Bento Specification

Revision 1.0d5

July 15, 1993

by Jed Harris and Ira Ruben

Apple Computer, Inc.

Bento: *n.* [Japanese] **1.** A box lunch or picnic lunch. **2.** A box or basket with multiple compartments, containing a collection of disparate elements arranged in an esthetically pleasing manner.

Table of Contents

Chapter 1: Introduction to Bento

Bento is a specification for storage and interchange of compound content. Bento defines two things: a format for containers of compound content, and an API to access these containers. Bento containers are used by application programs to store compound content that consists of multiple content objects. Bento is designed to be platform and content neutral, so that it provides a convenient container for transporting any type of compound content between multiple platforms. The Bento code corresponding to this specification currently runs on Macintosh, MSDOS, Microsoft Windows, OS/2 and several varieties of Unix.

This specification defines the format of data in a Bento container, and an API for writing and reading Bento containers. The Bento design is the result of several years of prototyping work, and extensive discussions between several major vendors.

If you have further questions about Bento or want to give us feedback, please contact Jed Harris via email. His address is jed@apple.com. We welcome your responses to Bento.

How to Read This Specification

This introduction and the following requirements chapter provide motivation for the Bento design. It is important to understand the requirements because Bento is a general container standard that addresses a very broad range of uses. Typically any single user of Bento encounters only a subset of these requirements, and it may be difficult to understand the motivation for some design decisions if you are thinking only of a subset of the requirements.

The API. may be easier to understand than the format, because it has some analogies to APIs of existing services such as file systems. However, it does deal with a number of subtle issues that file systems typically do not handle.

The Bento format is simple, but very flexible and recursive. The usage examples in chapter 8 should be helpful in understanding the concepts in the overview (chapter 6) and the description of the format (chapter 7).

Why We Need Bento _____

Increasingly, documents are made up of multiple content elements, such as text, tables, images, formatting information, mathematical equations, graphs, etc. Often content is created using one application and then included in documents created by other applications. Later, content elements may be copied out of a document and used in yet other documents. And so on.

Right now, applications typically have no way to exchange multiple content elements, unless they have a "private contract" about the format they will use. Furthermore, one application typically has no way to find the content elements in another application's document, so typically it cannot let a user copy the these content elements and reuse them. Finally, every application developer who wants to store multiple content elements in a document typically has to invent her own object storage mechanism.

These are the problems that Bento addresses. Bento provides a mechanism for storing content elements as objects. It defines a standard format for storing multiple different types of objects, and an API for writing out objects and reading them back in. Bento is designed to be as simple, flexible, and efficient as possible, so that it can be used to solve a wide variety of different content storage problems without requiring changes or extensions to the core specification. In turn, this will allow the widest number of applications to cooperate using Bento as a means for storing and exchanging their content elements.

More About Objects

For the purposes of this specification, we will consider an object as simply one or more hunks of data that "hang together" and that can be referenced by other data. Bento objects can be simple or complex, small (a few bytes) or large (up to 2^{64} bytes). Compared with objects in languages such as C++, Bento objects are typically larger and more complex, because they represent user meaningful content elements, rather than the atoms and molecules used to build this content.

For example, a sequence of bytes of data would not be an object, because we can only understand the bytes if we know how they will be used. A paragraph, an image, etc. can be an object if it contains enough information so that we know how to interpret it. Typically an object contains information about what kind of object it is, and some data, which provides the content of the object. In this specification, the information "about" the object is called <u>metadata</u>, and the content of the object is called its <u>value</u>.

What is a Container?

An object container is just some form of data storage or transmission (such as a file, a piece of RAM, or an inter-application message) that is used to hold one or more objects (both their metadata and their values). Bento containers are defined by a set of rules for storing multiple objects in a such a container, so that software that understands the rules can find the objects, figure out what kind of objects they are, and use them correctly. This is basically a simple idea, and Bento is a simple format. However, it was tricky to design, because it has to accomodate an enormous variety of different kinds of objects, different ways that applications want to use objects, and system considerations about how data can be stored.

Bento is intended to provide a container definition that can conveniently, efficiently, and reliably hold all the different kinds of objects that users and applications want to group together, store, and exchange. Bento does not define how any given object is structured internally, because there are already a very large number of different object formats around today, and we are still inventing new ones. Objects stored in an Bento container can have proprietary or standard formats, they can be designed to use the Bento mechanisms or they can be completely ignorant of the the existance of Bento.

How You Can Use Bento _____

To better understand why Bento is useful, let us look at some scenarios of how it can be used in specific cases:

Object Interchange

Let's say we are building a presentation. We want to take data in a spreadsheet, construct a 3D chart from it, and decorate it using a 3D modeling application. Then, we want to render it, animate it, and give it a sound track, titles, etc. As we go along, we want to keep track of the connection to the original information, so that if the data in the spreadsheet changes, we won't have to throw away all our work.

To do this, we will need to store the information created by our spreadsheet, charting application, modeling application, and animation application. In the process, we will be accumulating more and more different types of content elements, with lots of relationships between them. If the applications all agree to store their content elements as Bento objects, they can keep track of all the objects and their connections, even if (say) the animation application doesn't understand spreadsheet objects.

Document Storage

Continuing with this example, suppose we have all the objects set up in our animation, and we would like to save them in a file for future use. Furthermore, we would like to be able to find some of the objects and unpack them even if the animation application isn't around, so that (for example) we can reuse the spreadsheet data, or some of the 3D decorations we have cooked up. We can do this if the animation application stores the objects in a file formatted as a Bento container; then other applications can look inside the Bento file and find the objects they understand.

Enhanced Editing

Now suppose we come back and want to edit some of the objects in our animation. Applications can use two main strategies to edit a Bento document. First, they can read the document into memory, edit it there, and then write it out as a new Bento file. Some applications want to keep an entire document in memory, so this is a natural approach for them.

However, suppose our animation editor needs to handle documents too big to fit in available memory. In this case it can update the original Bento file, or create a new "delta" file. In either case, incremental changes are saved without modifying the existing data in the original file. This provides data integrity, so that the file won't be corrupted if the system crashes during an update. Only the minimal information required to record the changes is saved; for example, if four bytes in a 256K value are overwritten, only those four bytes plus some bookkeeping information are saved. However, in the current implementation of the Bento library, deleted or overwritten bytes are not recycled. Eventually, the application may wish to copy the file to compress out the unused space.

Compression and Other Data Transformations

Of course, the movie we are generating could be very large. However, Bento allows compression and decompression to be done on values. This mechanism is extensible, so a clever new compression scheme could be used that takes advantage of the characteristics of animation. Furthermore, the mechanism is transparent to the application, so the compression mechanism might be provided by a third party, and even added after the initial animation application has been written.

This same mechanism can be used for other types of data transformations as well. For example, the spreadsheet might contain sensitive financial information, so it might be encrypted, with only the users who need to be able to access it having a key.

References Across Containers

Maybe some of the backgrounds in our animation are scanned pictures that are available on a server in our department. These 24 bit scanned images are several megabytes each, and we don't want them on our local hard disk! So, we don't copy them into the animation; instead we put links to them in the file, and Bento knows that these are references, not the real objects, and transparently follows them to find the images in the server library. If the images are stored in Bento containers too, the references aren't limited just to a file, but can be to specific objects within a file.

If we use the Bento format for interchange between applications, when we use one of these images in one application, and then copy the result and paste it into a new application, all that has to be communicated is the reference, not the multi-megabyte image itself.

Finally, when we take our animation home to work on it in the evening, of course the references to the giant images don't work anymore, since we can't connect to the server. However, we can also save alternate representations in a Bento container. If we have saved low-resolution, 8 bit copies of the images, then, using Bento, the application can transparently use the version we do have when it can't find a complete image.

Multi-Media CD-ROM

Imagine that we want to build a CD-ROM using formatted text, video clips, sound, animation, "live" simulations, you name it. We want our different media related, so that when the user clicks on an animated object, it can play a sound or pop up a text description. Furthermore, the information on the CD-ROM has to be stored in a carefully crafted layout, so that when a low-cost machine is playing the CD-ROM, it can read and use different types of data (such as sound and video) "just in time", without a lot of memory to hold the data until it's time to use it, and without introducing annoying pauses while it searches the CD-ROM. This requires breaking large objects up into smaller chunks and interleaving them on the CD-ROM, while keeping track of which chunks belong to which objects.   Bento provides the flexibility required to do this interleaving, while leaving the decisions about the specific formats up to tools that understand the details of the media requirements.

In addition to just playing the CD-ROM, we want to have authoring tools that let users take the video, sound, animation, text, etc. objects on the CD-ROM and re-organize them, so that (for example) a teacher can turn reference material into customized courseware. The tools should be able to look at the objects without having to understand the complicated details of how they are laid out on the CD-ROM, and it should be able to create new objects in its own files that reference the old objects on the CD-ROM, just as though they were ordinary objects in a file. Bento allows these tools to view the interleaved objects as though they were stored normally in contiguous byte sequences.

Limitations of Bento

Bento is **not** intended to be used as a general purpose concurrent access database mechanism. Full support for such database style mechanisms would require much more complex libraries, and would conflict with the type of format control required for multimedia support (among other things). Thus it is not appropriate to try to use an Bento container as a general purpose concurrent access persistent object database.

Incremental update is possible using Bento. For example, an application managing a very large document might want to change the text in a paragraph, but avoid copying the entire document to make the change. Instead, it could update the contents of the paragraph, but leave the rest of the document unchanged. The effect of this update would be to make a new copy of the changed parts of the paragraph at the end of the document.

Eventually, the document would need to be copied to get rid of the "dead space" left behind by changes. Bento does not currently reuse such dead space. However, we have done analysis of the additional support required to reuse dead space, and it can be done without modifying the Bento storage format. This functionality is a candidate for Bento Version 2.0.

Scope of This Specification_____

This document (the current Bento specification) completely defines the format of a Bento container at the byte level. However, their are "higher level" issues that this document does not address. Perhaps we can best understand this approach if we look at any given container as being divided into a number of levels:

```
 _____
 \        Container format         /
  _____/
   \    Standard types and       /
    \      properties           /
     _____/
     /  Public types and      \
    /      properties          \
   /_____\
  /   Private types and          \
 /       properties               \
/_____\
```
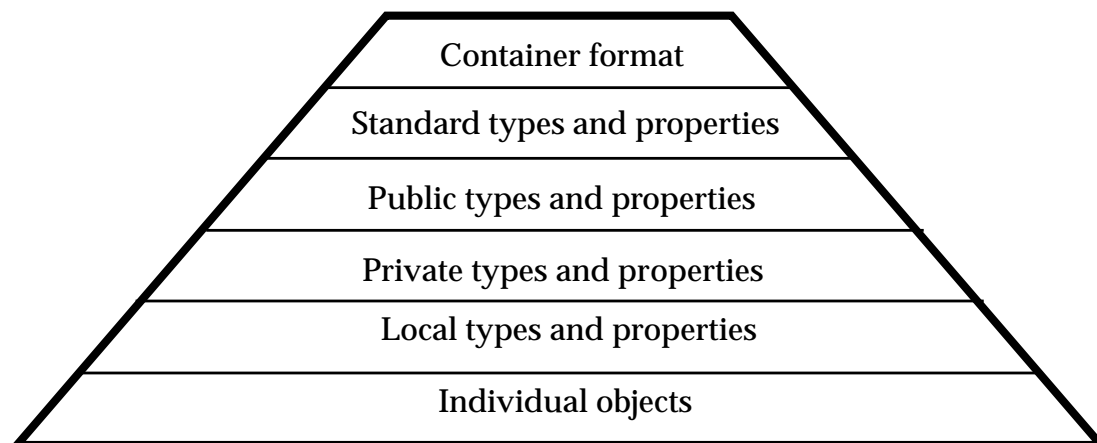

This specification only describes the top two levels of this diagram: the container format and the standard objects required by the Bento library.

There are several areas in which continuing work will be required if Bento is to achieve its potential:

First, in order to avoid unnecessary incompatibility, it is very desireable to have a strong basic set of public types. For example, we should have strings in a range of character sets, dates, integers, floating point numbers, etc. Data interchange will be greatly enhanced if we agree up front on these basic values. We are now in the process of listing and prioritizing the initial set of basic public types. We would appreciate feedback on the types you feel should be selected for initial standardization.

Second, addtional "layered" standards are needed in various areas. Such layered standards are being defined for multimedia data interchange (OMFI) and other types of content. Additional layered standards of interest include external reference mechanisms, such as system and file structure independent linking mechanisms.

Finally, to the extent that Bento is successful, many content standards will be defined within it. These standards will use Bento facilities, so they will be tied to Bento, and there will be a continuing need to register their types and properties, resolve potential conflicts between them, etc.

To support all of these areas, we plan to put in place a process for administering the Bento standard, and for registering and standardizing Bento types and properties.

Status of Bento

With the 1.0d5 release, we are approaching a final version of Bento 1.0. All previously planned changes have been completed. Products are currently shipping on the 1.0d5 version of the libraries, and we expect the format supported by the final version of the 1.0 libraries to be compatible with the 1.0d5 version.

Changes in Version 1.0d5

Only substantive changes made since version 1.0a4 have been marked with change bars.

The major changes are as follows:

New Table Of Contents Format

We adopted a new TOC format that will support 64 bit offsets, and that is also more compact and flexible than the old TOC.

A number of users of Bento expressed concern that value offsets in the original TOC were only 32 bits, because they need to support storage in which containers can be more than 2^{32} bytes long. We could clearly see that this would be a very serious issue for many users in a few years. Therefore, we concluded that Bento needed to e able to support 64 bit offsets.

As a result, we have developed a new TOC format that allows optional 64 bit offsets. The new format does not impose any overhead on those who do not use 64 bit offsets. This new TOC format is incompatible with the previous format. However, the 1.0d5 library detects and reads the previous format.

In addition to providing 64 bit offsets, the new format provides significantly reduced overhead. Previously, the minimum overhead for a value was 24 bytes in all cases; with the new format, it can be as low as five bytes in important cases.

Furthermore, the new format provides greater flexibility and considerably reduces the likelihood of further incompatible format changes in the future.

Reference Tracking

Copying or deleting structures of objects that refer to each other requires following references from one object to others. Furthermore, in general the references in a copied object must be adjusted.

In the previous API there was no standard way to find all references from a given value to other objects. This meant that we could not write standard utilities to perform deep copies, deep deletes, or to garbage collect a Bento container.

Furthermore, applications could not copy groups of objects without understanding the format of every value, because of the need to fix up references, and possible ID collisions when the copy was done to a different container.

We have made a pure extension to the API that solves these problems by allowing an application or utility to enumerate and fix up the references from any value, without understanding its format.

Error Reporting

Based on feedback from users, we changed the interface to the error handler to pass error numbers, rather than strings. This allows the error handler to more easily classify errors and determine whether they are recoverable or not.

This change only affects the API of the error handler.

Changes Planned

At this point only one minor change is definitely planned for Bento before it is final.

Add Force Alignment Call

We have been asked by the IMA, which is reviewing Bento for standardization as a the bottom layer of a multimedia interchange format, to provide some way to force alignment of values. While the Bento format currently permits this, the API provides no way to request it. This change only involves adding a single routine to the API and is completely upward compatible. It involves no changes to the format at all.

Changes Being Considered

At this point we have no other changes to Bento that are definitely planned. We are considering three possible changes. These would be the final changes to Bento (aside from any bug fixes) in Version 1.0.

We would very much like to hear from developers who either want these changes or would be significantly inconvenienced by them.

Merge Update List with Table of Contents

We are considering a change to the container format that would only affect updated containers. It would be an incompatible change to updated containers, although code would be provided to read containers in the 1.0d5 format.

This change would reduce the overhead of storing updates, and would increase the likelyhood that we could add storage reuse in Bento 2.0 without an incompatible format change.

Optional Explicit Error Return

A few developers have complained that they must run on some platforms that do not support nonlocal transfers of control (setjump / longjump), and that this makes the Bento error handling model difficult for them to use. In addition, some developers might prefer to have the API return error codes rather than call a handler that has to do a nonlocal control transfer.

We have designed a uniform change to the API that would address this problem. Developers who did not need it could ignore it, as we could provide a backward compatible cover for the modified API.

Error Severity Classification

Developers have asked us to provide a consistent way of determining the severity of an error—in particular, whether it is sensible to attempt to recover and continue, or whether it should be regarded as fatal. We are considering modifying the error handler API to include a severity indicator in addition to the error code itself.

Changes Rejected

In the 1.0d4 spec, we said that we were in the process of designing an accessor API for Bento. We decided not to do this because accessors could be implemented as a layer above the current API with little or no performance penalty and only a subset of Bento users wanted accessors anyway.

Chapter 2: Bento Requirements

These requirements have emerged as a consensus from consideration of a wide range of usage scenarios, and from detailed discussions with developers who will use Bento. At this point they are quite stable, but we anticipate some further evolution as we understand better how Bento will be used.

Primary Objectives

These objectives define the basic reasons for developing Bento, and the essential goals that it must meet.

Content neutral

The container format must store any kind of content efficiently, and must not be biased toward any particular kind of content. The container itself should be clearly separated from the content, so that the same software can be used to access the container regardless of what kind of content it holds.

Platform neutral

The container format must work well on any platform. It must not be biased toward any particular hardware or system software environment.

Concrete format

The container format must fully define the concrete details of the container (where the bits go), not **just** an API. The specifications must be clear and complete enough to ensure that developers who implement the format can interoperate.

Good for random access read

The container format must allow efficient access to individual objects, given their IDs, when a container is stored on a random access device (memory, disk, CD-ROM, etc.).

Allow for update in place

The format must allow for implementation of full storage reuse, so that the Bento format can be used as a native document format in a full range of applications.

Derived Requirements

Given the primary objectives above, we derived the following requirements, largely by considering usage scenarios from many different domains: multimedia, compound documents, cross-platform transport standards, application data interchange, and others. Each requirement below is supported by several usage scenarios.

Simple and small

The container format needs to be simple, to make it quick to implement, flexible, and "light weight" in terms of memory and processor support.

Standard API as well as format

Developers must be urged to access the format through an API, rather than creating their own reader/writer code from the format spec. This will make the format much more evolvable and will speed adoption.

The API must be able to support update in place usage, as well as write-once, to allow for a graceful transition to enhanced functionality when it becomes available.

External reference mechanism

The format must support external references in and out of the container. These references must be rebindable when a container is moved, and the mechanism for storing references must be extensible over time so that new reference semantics can be added.

Persistent references

Objects must have persistent identity so that they can be reliably referenced across many versions of a given file. If some objects are deleted and others are added, the old references should either (1) continue to successfully refer to the correct object, or (2) detectably fail to refer to any object.

Local generation of globally unique names

Names of externally meaningful entities (such as types, properties, methods, etc.) must be globally defined across all Bento containers, so that software that depends on understanding the entities can recognize them. However, at the same time, not all such names can be registered, because types, properties, methods, etc. will be developed by very large numbers of "casual" developers, such as spreadsheet and stackware programmers, in addition to "professional" developers who could register their work.

Thus, Bento must provide a mechanism for naming types, properties, etc. that can be used by large numbers of developers without registration, and still provide reliably unique names. Registration must still be supported for common names.

Object extensibility

Objects must be extensible. Applications must be able to attach new information to any object without disrupting other applications that don't understand the new information. Any information attached must have its own type, and must be able to be as large as any other part of the object. These requirements allow inspectable information to be associated with objects that would otherwise be opaque in some environments.

Support for Recursive Access and Embedded Streams

Values may themselves be containers or streams. Recursive access to embedded containers (either Bento or other formats) must be convenient and must support existing container usage. Access to streams must be convenient and must support existing stream usage.

Encoding defined per value

Objects created on different platforms, using different encoding standards, must be able to coexist within a standard container. This implies that byte-ordering, etc. must be defined at the value format level.

Multiple data formats for a single object

Sometimes an object used in multiple environments will need to be stored in multiple formats. The Mac clipboard and edition files are current examples. OLE is also an example: an embedded object is typically stored in two or three formats. In a multi-platform situation, in which a file is on a file server, being accessed from different environments, even more extreme usage scenarios along this line are likely to emerge.

Consistency checking

Applications using a container must have some way to tell which objects and values were updated when. This is important so that when we have multiple formats, related properties, etc. an application can at least detect which ones are consistent, and which ones may be inconsistent.

Maximum possible layout flexibility

The container format should impose the least possible constraint on the structure and location of values, allowing them to be stored in any order, broken up into pieces and interleaved, even nested inside of other values. This is necessary so that any bizarre requirements on layout can be met, including interleaving required for real-time playing of multi-media, constraints imposed by other standards, etc.

Out-of-line metadata

The container format must support values with no in-line type, size etc. This is necessary so that the format bytes are not in the way during reading some stream formats (including interleaved multi-media).

Inspectable references from each object

An application must be able to identify all objects that are referenced by a given object, **without** knowing the details of the value formats in the first object. Such a mechanism is required to enable applications or utilities to copy object structures,, delete object structures without dangling references, etc. without completely understanding all the formats of all the objects.

Remappable references from each object

An application must be able to change the target of references in a given object, without knowing the details of the value formats. Such a mechanism allows an application or utility to fix up the references when copying an object structure. This fixup is required to make the references point correctly to the copied objects, rather than the original objects. It is also useful in a variety of other contexts.

Chapter 3: Design Overview

This chapter provides an overview of the Bento design. It describes it more from the API perspective than the format perspective. However, most of the concepts also apply to the format level. In some respects the format is simpler than the API, but it difficult to understand without first understanding the API functionality it is intended to support.

Bento Data Model

The easiest way to begin understanding the Bento design is probably to review the entities that the API manipulates.

Primary Entities

The most important entities in the Bento design are containers, objects, properties, values, and types. Their relationship is displayed in the following diagram:

Container

Property p

Property k

Object A

Property n

Property s

Value of type x
Value of type y
Value of type u
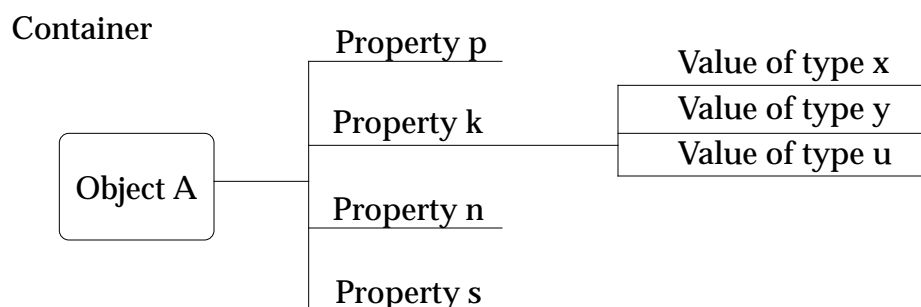
Every object is in some container. An object consists of a set of properties. The properties are not in any particular order.   Each property consists of a set of values with distinct types. The values are not in any particular order. Every object must have at least one property, and that property must have at least one value. Each value consists of a variable length sequence of bytes.

Now let us look at these primary entities in more detail.

Containers

All Bento objects are stored in containers. Bento knows very little about a container beyond the objects in it. However, the container always contains a distingushed object, and applications can add arbitrary properties to that object, so applications can specify further information about the container if they wish.

Containers are often files, but they can also be many other forms of storage. For example, in various applications developers already support the following types of containers: blocks of memory, the clipboard, network messages, and Bento values. Undoubtedly other types of containers will be useful as well.

Objects

Each Bento object has a persistent ID which is unique within its container. Other than that, objects don't really exist independent of their properties. An object contains no information beyond what is stored in its properties.

Properties

A property defines a role for a value. Properties are like field names in a record or struct, with two differences. First, properties can be added freely to an object, so an application should never assume an object only has the properties it knows about. Second, property names are globally unique, so that they can never collide when various different applications add properties to the same object. This also means that the same property name always means the same thing, no matter what object it is in. Properties are distinct from types, just as field names are distinct from the data type of the field.

For example, different properties of an object might indicate the name of an object, the author of the object, a comment, a copyright notice, etc These different properties could all have values of the same type: string.

Conversely, a property indicating the date created might have a string, Julian day, or OSI standard date representation. These different formats would not be indicated by the property, but by the type (see below).

Values

Values are where the data is actually stored. The data for a value can be stored anywhere in a container. In fact, it can be broken up into any number of separate pieces, and the pieces can be stored anywhere. (See the discussion of value segments below.)

Each value may range in size from 0 bytes to 2^{64} bytes (if you have that much storage). The overhead per value varies depending on the circumstances. For an object with a single value, the typical overhead will be 21 bytes. For a small value which is one of several values associated with a property, the overhead can be as low as five bytes.

Types

The type of a value describes the format of that value. Types record the structure of a value, whether it is compressed, what its byte ordering is, etc. Bento provides an **open-ended** mechanism, so that types can be extended to include whatever metadata is required.

To continue the example above, the type of a string value would indicate the alphabet, whether it was null terminated, and possibly other information (such as the intended language). It might also indicate that the string was stored in a compressed form, and would indicate the compression technique, and the dictionary if one was required. If the string used multi-byte characters, and the byte-ordering was not defined by the alphabet, the type would indicate the byte-ordering within the characters.

Bento defines an inheritance mechanism to make building complex types like this efficient. The structure of types is tied into the mechanism for accessing values, so that the type associated with a value causes the appropriate code to be invoked to access the value, decompress it, byte-swap it, etc. The specific mechanism for doing this is discussed in "Dynamic Values" below.

Secondary Entities

There are several additional entities that play supporting roles in the Bento design. These entities are important to fully understand how Bento works, but they do not significantly change the picture given above.

Type and property descriptions

Each property associated with a value is a reference to a property description. Similarly, the type is a reference to a type description. These type and property descriptions are objects, and their IDs are drawn from the same name-space as other object IDs.

Many type and property descriptions will simply consist of the globally unique name of the type or property. To continue the example above further, the type of a string of 7-bit ASCII, not compressed or otherwise transformed, would simply be described by a globally unique name. This would allow applications to recognize the type.

References to type and property descriptions are distinct from references to ordinary objects in the API to allow language type checking to catch errors in the manipulation of type and property references. However, type and property references can still be passed to the object and value operations, so that value manipulation can be done on types and properties as well as normal objects.

Globally unique names

Globally unique names public or private indentifiers in a format defined by the ISO 9070 standard. They are simply strings written in a subset of 7 bit ASCII. They begin with a name that is assigned by a naming authority designated by ISO (companies can easily register as naming authorities). After this come additional segments, as determined by the naming authority, each of which is unique in the context of the previous segments.

The most common globally unique names will be generated by system vendors or commercial application developers, and may be registered. However, in many cases names will be generated by vertical application developers to record their local types and properties. To meet this need, the naming rules allow for local creation of unregistered unique names, for example by using a product serial number as one of the name segments.

Note that while these names are in 7-bit ASCII, and can easily be printed, they are not designed to be meaningful to end-users. No doubt developers will find reading the specific names useful, but messages to users should be provided in terms of names explicitly designed for user consumption. Such names can easily be provided in type and property descriptions.

IDs

Each object is assigned a persistent ID that is unique within the container in which the object is created. These IDs are never reused once they have been assigned, so even if an object is deleted, its ID will never be reassigned.

These IDs are obviously essential to the functioning of the Bento format, but they do not appear directly in the API. The only points at which an application actually deals with anything corresponding to an ID is when it needs to store an object reference into a value, or find the object corresponding to a reference retrieved from a value. Even in this case, however, the API does not give the application direct access to an object ID, but only to a token that corresponds to the ID in the context of that particular value. This hiding of actual IDs is necessary to allow for reference tracking.

Refnums

In the API types, properties, and objects are referred to using opaque refnums ("magic cookies") provided by the API. The refnums are much more convenient to use than IDs because they are unique within the session, while an ID would need to be used together with a container reference. Since they are opaque, they allow implementations of the API that support caching schemes in which only portions of the container metadata are in memory at any given time.

Refnums have no persistent meaning, so they cannot be stored in values as references to other values. The tokens provided by the reference calls must always be used for persistent references.

Dynamic values

As mentioned above in the discussion on "Types", a Bento value can be compressed, encrypted, byte-swapped, etc. during I/O. Furthermore, these transformations can be composed together.

In addition to data transformation, the same mechanism also supports I/O redirection. In this case a value actually stored in a container is a description of how to find the data, rather than the data itself. Such descriptions can be as simple as references to files, or to objects in another container, or as complex as queries that cause data to be retrieved from a database.

Both I/O transformations and I/O redirection are carried out implicitly by the Bento library, using handlers determined by the type of the value (see Handlers, below). These handlers are attached to temporary entities called dynamic values created by the library. Dynamic values are never visible to the application, and have no persistent meaning.

Value segments

To support interleaving and other uses that require breaking a value up into pieces, Bento allows a value to consist of multiple segments stored at different locations in the container. These segments are not visible at the API, which glues them together to create a single stream of bytes.

The API also takes advantage of value segments to represent insertions, deletions, and overwrites of contiguous bytes in a value. This allows Bento to represent these operations directly in recording updates, rather than having to create a new copy of the value.

Handlers

Bento makes use of dynamically linked handlers supplied by the execution environment for two reasons:

- •   Portability. Use of handlers means that the Bento library is almost trivially portable, since all the system dependencies are in the handlers.
- •   Extensibility. The Bento library is designed to be easily extended by writing new handlers. The hander interfaces are carefully designed to provide cleanly encapsulated abstractions.

There are three types of handlers:

Session handlers

Certain operations are global to the session as a whole. These include allocating and deallocating memory, and reporting errors. These handlers provide portability, but don't do much for extensibility.

Container handlers

Actual I/O to containers is always done using container handlers, to provide platform independence. The many different types of containers mentioned in the first section are not actually implemented in the Bento library. Instead, the library simply calls different types of handlers, all of which provide the same interface. These handlers map I/O to the underlying storage in a way that depends on the container type. Container handlers basically provide a stream I/O interface to the container storage.

Clearly, container handlers provide portability, by insulating the library from the I/O mechanisms. They also provide extensibility, allowing new types of containers to be defined. Less obviously, they allow extension of the I/O mechanism, such as the addition of transparent buffering without modification of the library. Finally, they support portability between machines with different byte order and even different sizes of bytes.

Value handlers

Both I/O transformations and indirect values are implemented by value handlers, and these handlers are determined by the type of the value. New handlers to carry out new types of data transformations or support new types of indirect values can be written at any time.   These handlers are invoked entirely by the library.   The accessing application does not need to know that it is using handlers to access the value.

Value handlers are provided mainly for extensibility. However in some cases, such as platform-specific interpretation of references, they also support portability.

Chapter 4: API Definition

Changes Since Version 1.0a4

Aside from the introduction of the additional reference manipulation routines, the API changes are essentially refinements of the 1.0a4 spec.

Minor changes to the API are not listed here, but are marked with change bars in the body of the chapter. Purely expository changes are not marked at all.

Addition of Reference Routines

Four reference manipulation routines have been added to the API: `CMSetReference`, `CMDeleteReference`, `CMCountReferences`, and `CMGetNextReference`. In addition, for consistency, `CMGetReferenceData` has been renamed `CMNewReference`; its interface is unchanged.

Change to the Error Handler Prototype

The error handler prototype has been changed to take an error number and a variable number of strings rather than a single error string. This allows easier classification of errors reported to the handler.

Change Generation Number to 32 Bits

CMGeneration has been changed to 16 bits to 32 bits. A number of Bento reviewers had expressed a concern that a significant number of Bento containers might have lifetimes of more than 2^{16} generations. This change will generally impose no additional TOC overhead because of the new TOC format.

Container Encoding Control Eliminated

The encoding argument has been eliminated from CMOpenNewContainer. All container TOCs will have little-endian encoding.

AbortSession and AbortContainer Calls Added

Calls to abort a session or a container cleanly after unrecoverable errors were added.

Design Comments

Portability

The actual API headers are annotated with macros to support the various declarators needed for different systems and especially different C compilers. These annotations have been removed from the API as presented in this chapter of the specification.

We have carried out extensive testing to make sure that Bento will run in as many environments as possible.

Declaration Style

Declarations are deliberately made in a platform-independent manner. A mapping from the declarations as given to a specific platform will be required for each implementation.

Names specific to the API are prefixed with "CM" (Container Manager).

<u>Error Reporting</u>

The API calls an error handler provided in the initialization call to report errors. This handler could do a longjump, or it could use a more sophisticated error reporting scheme.

<u>Error Codes</u>

The error codes returned by the operations are defined in Appendix C. This list will continue to grow with extensions to the design. The list of errors possible from each operation is not yet included in the spec.

Types and Constants _____

Low level basic types

```
typedef char        CHAR;

typedef unsigned char        UCHAR;
```

Signed and unsigned1-byte values.

```
typedef short       SHORT;

typedef unsigned short       USHORT;
```

Signed and unsigned2-byte values.

```
typedef long        LONG;

typedef unsigned long        ULONG;
```

Signed and unsigned 4-byte values.

Types

All Container Manager types are defined here with the intent to aid in enforcing compiler type checking. Note that "refNums" for session data, container control blocks, values, and objects are uniquely typed to strictly enforce type checking of those entities. These types are defined as pointers derived from an "incomplete type", i.e. structs. The structs are **not** defined. In ANSI C, an "incomplete type" need not be defined.

```
typedef struct                    CMSession_ *CMSession;
```

Pointer to session (task) data.

```
typedef struct                    CMContainer_ *CMContainer;
```

"RefNum" for containers.

```
typedef struct                    CMObject_ *CMObject;
```

"RefNum" for objects.

```
typedef CMObject                  CMProperty;
```

"RefNum" for property description objects.

```
typedef CMObject                  CMType;
```

"RefNum" for type description objects.

```
typedef struct                           CMValue_ *CMValue;
```
"RefNum" for values.

```
typedef CHAR                             *CMOpenMode;
```
Handler open mode string pointers.

```
typedef CHAR                             *CMGlobalName;
```
Global unique name pointers.

```
typedef CHAR                             *CMErrorString;
```
Error message string pointers.

```
typedef CM_CHAR                           *CMMetaData;
```
Type metadata string pointers.

```
typedef void                             *CMRefCon;
```
Reference constants ("refCon"s).

```
typedef void                              *CMPtr;
```
Arbitrary data pointers.

```
typedef UCHAR                            *CMMagicBytes;
```
Magic byte pointers.

```
typedef CM_UCHAR                         *CMDataPacket;
```
"New value" handler data packets.

```
typedef CM_UCHAR                         *CMDataBuffer;
```
Ptr to data buffer for handlers.

```
typedef CM_UCHAR                         *CMPrivateData;
```
Ptr to private CM data for handlers.

```
typedef CM_UCHAR                         CMReference[4];
```
Referenced object data pointers.

```
typedef UCHAR                            CMSeekMode;
```
Container "fseek()" handler modes.

```
typedef UCHAR                            CMBoolean;
```
Boolean funct. results (0==>false).

```
typedef USHORT                           CMContainerUseMode;
```
Container open use mode flags.

```
typedef USHORT                           CMContainerFlags;
```
Container label flags.

```
typedef CM_USHORT                    CMContainerModeFlags;
```
Container open mode flags.
```
typedef USHORT                       CMEofStatus;
```
"feof()" handler result statusc
```
typedef CM_LONG                      CMErrorNbr;
```
Error handler error numbers.
```
typedef ULONG                        CMGeneration;
```
Container generation numbers.
```
typedef ULONG                        CMSize;
```
Sizes
```
typedef ULONG                        CMCount;
```
Amounts or counts.
```
typedef void                         *CMPtr;
```
Arbitrary data pointers.
```
typedef void                         ( *CMHandlerAddr)();
```
Handler address pointers.
```
typedef  CMHandlerAddr ( *CMMetaHandler)
                       (CMType,
                        const CMGlobalName);
```
Metahandler prototype.

Constants

The following flags are passed to CMOpen[New]Container(). They modify the open in the indicated ways.   Note that kCMReading, kCMWriting, kCMUpdating are also returned from CMGetContainerInfo() to indicate the mode of the container, i.e. it was opened for reading, writing, or updating.

```
const CMContainerUseMode             kCMReading = 0x0001
```
Container was opened for reading.
```
const  CMContainerUseMode            kCMWriting = 0x0002
```
Container was opened for writing.
```
const CMContainerUseMode             kCMReuseFreeSpace = 0x0004
```
Try to reuse freed space.
```
const CMContainerUseMode             kCMUpdateByAppend = 0x0008
```
Open container for update-by-append.
```
const CMContainerUseMode             kCMUpdateTarget = 0x0010
```

Open container for updating target.

```
const CMContainerUseMode              kCMConverting = 0x0020;
```

Open a container for "converting".

The following flags are options to the "seek" I/O handler.

```
const CMSeekMode                      kCMSeekSet = 0x00;
```

"fseek()" handler mode (pos).

```
const CMSeekMode                      kCMSeekCurrent= 0x01;
```

"fseek()" handler mode (curr+pos).

```
const CMSeekMode                      kCMSeekEnd = 0x02;
```

"fseek()" handler mode (end+pos).

Operation Definitions _____

Session Operations

The session is an explicit object created by initializing the library. It represents private Container Manager data that is global to all open containers. The intent is that this data is unique to the currently running session (or task). The caller may extend this data to include his or her own special per-session information.

```
CMSession  CMStartSession(CMMetaHandler metaHandler,
                    CMRefCon sessionRefCon)
```

This call is used for all global initialization of the Container Manager. It **must** be called before any other Container Manager routine and should only be called once. If not, every API routine will try to exit without doing anything.

An anonymous non-NULL pointer is returned if initialization is successful. NULL is returned for failure unless the error reporter (discussed below) aborts execution.

This routine takes as its main parameter the address of a metahandler. This meta-handler must define operations for error handling, memory allocation, and memory deal-location. The interface to the metahandler and to the three specific handlers is documented in Appendix B.

In addition the caller can pass a "reference constant" (refCon) as the last parameter to this routine. It is saved in the session data. The refCon is not used by the API and can be anything, but usually it will be a pointer to the caller's own session data.

```
void  CMEndSession(CMSession sessionData,
                    CMBoolean closeOpenContainers)
```

This should be called as the **last** call to the Container Manager. It frees the space allocated for the session by CMStartSession() and optionally calls CMCloseContainer() on all remaining open containers.

The sessionData specifies the session data pointer returned from `CMStartSession()`. If `closeOpenContainers` is passed as 0 (i.e., "false"), then an error is reported for each container that has **not** been explicitly closed by `CMCloseContainer()`. If true (non-zero) is specified, then the Container Manager will call `CMCloseContainer()` for you for each remaining open container.

No further calls should be done once this routine is called. All memory occupied by the containers, as well as the session itself are freed.

```
void  CMAbortSession(CMSession sessionData);
```

This is basically a `CMAbortContainer()` for all currently open containers followed by a `CMEndSession()`. This routine **will** return to its caller. It is up to the user to actually abort execution if that is required. This call is intended to be used to abort the session from unrecoverable errors.

All containers are closed without further writing to those containers, i.e., as if all containers were opened for reading even when opened for writing. All memory allocated by all the container data structures are freed (if possible) and the container close handlers called to physically close the containers. All dynamic values currently in use are released in an attempt to allow them to properly clean up any files and memory allocated by their handlers. No further API calls should be done.

```
CMRefCon  CMGetSessionRefCon(CMContainer container)
```

This routine can be used to get at the user's session refCon saved as part of the session data created by `CMStartSession()`. The session data is "tied" to each container created by `CMOpen[New]Container()`. Thus typically the refCon will be accessed via a container refNum.

```
void  CMSetSessionRefCon(CMContainer container,
                CMRefCon refCon)
```

This routine may be called to change the user's session refCon associated with the session data.

```
CMHandlerAddr  CMSetMetaHandler(const CMSession sessionData,
                const CMGlobalName typeName,
                CMMetaHandler metaHandler)
```

This routine records the association of Global Names with their metahandlers.

The designated metahandler will be associated with the `typeName`. The previous metahandler for this type name, if any, is returned. If there was no previous metahandler defined, `NULL` is returned. The association between handlers and type names is global within a session, rather than specific to a given container.

A metahandler will be called whenever Bento or the application needs to find out how to perform a given operation on a container or value of this type. The metahandler can define specific handlers for any number of different operations, potentially with completely different interfaces. The interface to the metahandler is documented in the section on handler interfaces.

This routine must be used to associate a type name with a metahandler before `CMOpen[New]Container()` is called, so that the Container Manager can find the appropriate metahandler for the container.

```
CMHandlerAddr   CMGetMetaHandler(const CMSession sessionData,
                      const CMGlobalName typeName)
```

This function searches the metaHandler symbol table for the specified `typeName` and returns the associated metahandler address.   If no metahandler is associated with that type name, it returns `NULL`.

```
CMHandlerAddr CMGetOperation(CMType targetType,
                      const CMGlobalName operationType);
```

This routine takes a `targetType` which has a globally unique name and uses that name to find a metahandler. The metahandler, in turn, is called to get the handler routine address for the specified `operationType`. The function returns the resulting address.

Metahandler proc addresses are given to the Container Manager by calls to `CMSet-MetaHandler`. The global name for the input `targetType` is treated as the `typeName` to find the metahandler.

See Appendix B for more information on the hander mechanism.

Container Operations

Containers (files and blocks of memory) are always accessed through handlers, to provide platform independence and support nested containers. Handlers are responsible for creating a container if necessary, opening and closing it, managing stream I/O to it, and reading and writing the container label (which provides such information as the location of the Table of Contents). The interfaces to container handlers are documented in the Appendix B.

The types of storage that can be used as containers are limited only by the types of handlers available.

```
CMContainer   CMOpenContainer(CMSession sessionData,
                      CMRefCon attributes,
                      const CMGlobalName typeName,
                      CMContainerUseMode useFlags);
```

This operation opens an existing Bento container.

The `attributes` must designate management structures for the container storage. This `attributes` argument is not examined by Bento, but is simply passed to the appropriate handler interfaces.   It is intended to provide the information necessary for the handlers to locate a specific container. Thus `attributes` serves as a communication channel between the application and the Open handler. In its simplest form for a container file it would be a pathname. For an embedded container, it would be the parent value (CMValue), corresponding to the embedded container.

The `typeName` is used to find a metahandler defined for that same typeName. The metahandler, in turn, defines the handlers for the container and thus knows how to get at the physical container. These handers must understand the `attributes` provided.

The useFlags must be 0 or `kCMReuseFreeSpace`. 0 implies that the container is to be open for reading only. No writes may be done. If `kCMReuseFreeSpace` is specified, than **both** reading and writing may be done to update the container. Free space from deleted data will be reused and overwrites of existing data may be done to change it (subject to the container label flags, see below).

A container refnum is returned.

Note that an individual value can be opened as an embedded container. Through the attributes, the value is passed to the handlers. This value must be typed as an embedded container value. Embedded containers can have embedded containers which can also be opened and read. The effect is that a tree of nested containers can be opened and read without restriction. However, when a `CMCloseContainer()` is done on a parent container, all of its descendents will also be closed.

```
CMContainer    CMOpenNewContainer(CMSession sessionData,
                   CMRefCon attributes,
                   const CMGlobalName typeName,
                   CMContainerUseMode useFlags,
                   CMGeneration generation,
                   CMContainerFlags containerFlags,
                   ...);
```

This operation opens a new Bento container for writing. This is similar to opening for reading (see documentation above) except that here a new and empty container is opened. A minimum TOC is created along with the special TOC object 1 with its seed and offset properties.

The resulting container can be updated.

In addition to `kCMReuseFreeSpace`, the useFlags may be 0, `kCMConverting`, `kCMUpdateByAppend`, or `kCMUpdateTarget`.

`generation` is the generation number of the container; it must be ≥ 1. If this container is a copy of a previous container, the generation number should be 1 greater than the generation number of the previous container.

`containerFlags` is the flag value that will be stored in the container label. No container flags are currently defined.

If the `kCMConverting` flag is set in `CMContainerUseMode`, the physical container is assumed to already contain a sequence of bytes that the caller wants to convert to container format. The application uses `CMDefineValueData()` to create values for objects in the bytes. All new stuff, including the TOC is written at the end of the existing stuff. Bento will not modify the existing data.

If the `kCMUpdateByAppend` or `kCMUpdateTarget` flags are set, all updates to a "target" container are recorded in the container being opened. Future opens of this container, with `CMOpenContainer()` will apply the updates to the target to bring it "up-to-date" while it is open.

If `kCMUpdateByAppend` is specified, then the container is opened for update-by-append. All updates are appended to the existing container and an additional TOC is layered on to the end of the container when closed. Each time the container is opened and then closed for update-by-append, the new updates and a new TOC are appended. Whenever such a container is opened (in any mode), all the updates are applied appropriately to the original container.

Using `kCMUpdateTarget` is similar to `kCMUpdateByAppend`, but the updates are recorded in a new container.

In both cases the "target" container is specified in a type-dependent way, using the `CMRefCon` and the "`...`" parameters passed to `CMOpenNewContainer()`. These parameters are interpreted in exactly the same way as the corresponding parameters of `CMNewValue()`; see the documentation of `CMNewValue()` for further details on the "`...`" parameters.

A container refnum is returned.

Just as in reading, any number of embedded containers can be opened. Also embedded containers can be opened within embedded containers to any depth. The effect is that a tree of nested containers can be opened and written without restriction. However, when a `CMCloseContainer` is done on a parent container, all of its descendents will also be closed.

It is an error to call `CMOpenNewContainer` with a value that belongs to a container that is not updatable, since that call would create an embedded container open for writing.

```
void CMGetContainerInfo(const CMContainer container,
                        CMGeneration *generation,
                        CMContainerFlags *containerFlags,
                        CMGlobalName typeName);
```

The corresponding values for the designated container are returned. `NULL` may be passed for any reference; in that case the corresponding value is not returned.

```
CMSession  CMGetSession(CMContainer container)
```

The session global data pointer returned from CMStartSession() is passed to most of the handler routines defined in this file. This routine is provided to make it easier to retrieve the pointer as a function of the container refNum.

```
VOID CMCloseContainer(CMContainer container);
```

If the container was open for writing, all I/O to the designated container is completed, and the table of contents and label are built and written out.

This call closes the specified container and **all** its currently opened embedded containers (if any).

This call destroys the association between the container refnum and the designated container. On return the specifed container refNum and all the others corresponding to the embedded containers are invalid. All memory associated with a container's data structures is freed. After this call the container refNum may be returned by a subsequent `CMOpenContainer` call, designating another container.

```
VOID CMAbortContainer(CMconst_CMContainer container);
```

The container is closed without further writing to the container, i.e., as if it were opened for reading even when opened for writing. This is intended to be used to abort processing of the container from unrecoverable errors.

All memory allocated by the container data structures is freed (if possible) and the container close handler called to physically close the container. All dynamic values currently in use are released in an attempt to allow them to properly clean up any files and memory allocated by their handlers. No further API calls should be done on the container as it will be closed upon return.

Note, this routine will return to its caller. It is up to the user to actually abort execution if that is required.

Type and Property Operations

All types and properties must be registered before they can be used. The operations behave the same on standard types and properties as on normal types and properties. However, standard types and properties will not actually be given TOC entries for their descriptions just because they are registered. If additional, non-standard properties are added to the description of a standard type or property, they will be stored.

The refnum returned from registration can be used in exactly the same manner as an object refnum in the object and value operations.

Types and properties may be registered more than once; the refnum returned from all the different registrations of the same type is the same. Identity of types is defined by string equality of their names.

```
CMType CMRegisterType(CMContainer targetContainer,
                    const CMGlobalName name);
```

The designated type is registered in the designated container, and a refnum for it is returned. If a type with that name already exists, the refNum for it is returned.

Standard types may be registered, but this is not required.

```
CMProperty CMRegisterProperty(CMContainer targetContainer,
                    const CMGlobalName name);
```

The designated property is registered in the designated container, and a refnum for it is returned. If a property with that name already exists, the refNum for it is returned.

Standard properties may be registered, but this is not required.

```
CMBoolean CMIsType(CMObject theObject);

CMBoolean CMIsProperty(CMObject theObject);
```

These operations test the designated object and return non-zero if it is a type descrip-tion or a property description, respectively, otherwise 0.

```
CMType CMGetNextType(CMContainer targetContainer,
                     CM CMType currType);
```

A refnum for the next type registered in the same container is returned. `currType` is generally a refNum previously returned from this routine. Successive calls to this routine will thus yield all the type descriptions in the container.

Types are returned in order of increasing ID. If there are no larger type IDs regis-tered, `NULL` is returned. To begin the iteration, pass `NULL` as the type refnum.

```
CMProperty CMGetNextProperty(CMContainer targetContainer,
                     CMProperty currProperty);
```

A refun for the next property registered in the same container is returned. `currProperty` is generally a refNum previously returned from this routine. Successive calls to this routine will thus yield all the property descriptions in the container.

Properties are returned in order of increasing ID. If there are no larger property IDs registered, `NULL` is retured. To begin the iteration, pass `NULL` as the property refnum.

```
CMCount  CMAddBaseType(CMType type,
                     CMType baseType)
```

This routine defines base types for a given type so that layered dynamic values can be created. Base types essentially provide type inheritance. See the chapter on Types and Dynamic Values for a full description of how base types are used.

A base type is added to the specified type. For each call to `CMAddBaseType()` for the type a new base type is recorded. They are recorded in the order of the calls. The total number of base types recorded for the type is retuned. 0 is returned if there is an error and the error reporter returns.

It is currently an error to attempt to add the **same** base type more than once to the type.

```
CMCount  CMRemoveBaseType(CMType type,
                     CMType baseType)
```

The specified base type previously added to the specifed type by `CMAddBase-Type()` is removed. If `NULL` is specified as the baseType, **all** base types are removed. The number of base types remaining for the type is returned.

Note, no error is reported if the specified base type is not present or the type has no base types.

Object Operations

```
CMObject CMNewObject(CMContainer targetContainer);
```

A refnum to a new object in the designated container is returned. At this point the object has nothing but an identity.

```
CMObject CMGetNextObject(CMContainer targetContainer,
                    CMObject currObject);
```

A refum for the next object defined in the same container is returned. currObject is generally a refNum previously returned from this routine. Successive calls to this routine will thus yield all the objects in the container.

Objects are returned in order of increasing ID. If there are no larger object IDs defined, NULL is returned.   To begin the iteration, pass NULL as the object refnum.

Since type and property descriptions are objects, they will be returned in sequence as they are encountered. Only objects in the current container will be returned, not objects in any base containers.

```
CMProperty CMGetNextObjectProperty(CMObject theObject,
                    CMProperty currProperty);
```

A refnum for the next property defined for this object is returned. currProperty is generally a refNum previously returned from this routine. Successive calls to this routine will thus yield all the properties for the given object.

This routine returns the refNum for the next property defined for the given object. If there are no more properties defined for this object, NULL is returned. If currProperty is NULL, the refNum for the first property for the object is returned.

```
CMObject CMGetNextObjectWithProperty(CMContainer targetContainer,
                    CMObject currObject,
                    CMProperty property)
```

This routine returns the refNum for the next object in the container that has the given property. currObject is generally a refNum previously returned from this routine. Successive calls to this routine will thus yield all the objects with the given property.

If currObject is NULL, the search starts with the first object in the container. If there is no next object with the given property, NULL is returned.

```
CMContainer CMGetObjectContainer(CMObject theObject);
```

The container of the designated object is returned.

```
CMGlobalName CMGetGlobalName(CMObject theObject);
```

The name of the designated object is returned. This operation is typically used on types and properties, but it can be applied to any object with a Globally Unique Name property. NULL is returned if the object does not have a Globally Unique Name.

```
CMRefCon  CMGetObjectRefCon(CMObject theObject)
```

This routine returns the user's "refCon" (reference constant) that s/he may associate with any object refNum (i.e., a `CMObject`). The refCon is a `CM_ULONG` that the user may use in any way. It is not touched by the API except to initialize it to 0 when the object is read into memory.

Note that the refCon is **not** preserved across closed containers, i.e., it is not saved in the TOC.

```
void  CMSetObjectRefCon(CMObject theObject,
                        CMRefCon refCon)
```

This routine is used to set the user's "refCon" (reference constant) to be assoicated with an object. The refCon is a `CM_ULONG` that the user may use in any way. It is not touched by the API.

Note that the refCon is **not** preserved across closed containers, i.e., it is not saved in the TOC.

```
VOID CMDeleteObject(CMObject theObject);
```

The specified object and all its properties and values are deleted. It is an error to use the object refnum after this call has been made.

A deleted object will be treated by all Bento operations as though it does not exist. For example, it will not be found by `CMGetNextObject`, etc.

Objects containing values that are currently open embedded containers cannot be deleted. Also, some objects created and used in the management of the TOC itself cannot be deleted.

```
VOID CMDeleteObjectProperty(CMObject theObject,
                        CMProperty theProperty);
```

The designated object property is deleted along with all of its values.. It is an error to use the refnum of any value of this property of this object after this call has been made.

A deleted object property will be treated by all Bento operations as though it does not exist. For example, it will not be found by `CMGetNextObjectProperty`, etc.

Object properties containing values that are currently open embedded containers cannot be deleted. Also, some object properties created and used in the management of the TOC itself cannot be deleted.

```
VOID CMReleaseObject(CMObject theObject);
```

The association between the object refnum and the designated object is destroyed. After this call the refnum is invalid and may be returned from one of the object calls to designate another object.

This call is also used to destroy the association between properties and types and their associated refnums.

Value Operations

All the I/O calls in the current API do I/O to or from a buffer provided by the application.

The generation number can be set, but it is not passed in the `NewValue` call. NewValue defaults the generation number to the current generation value for the container. Normally this is what is wanted.

```
CMCount CMCountValues(CMObject object,
                      CMProperty property,
                      CMType type);
```

A property for an object can be defined to have more than one value. This routine returns the number of values for the specified property belonging to the specified object.

If the type is specified as `NULL`, the total number of values for the object's property is returned. If the type is not `NULL`, 1 is returned if a value of that type is present (because there can be a maximum of one value of that type), and 0 otherwise. If the property is not defined for the object, 0 is always returned.

```
CMValue CMUseValue(CMObject object,
                   CMProperty property,
                   CMType type);
```

This routine is used to get the refNum for the value of an object's property of the given type. `NULL` is returned if the value does not exist, or if or the object does not contain the property. If the type of the value corresponds to a global type name that has an associated "use value" handler, or if its base types (if any) have associated "use value" handlers, a the refnum returned will refer to a dynamic value rather than the base value. (Normally, an application will never be aware of this difference.)

If the value is typed as an embedded container, then this refnum can be passed to `CMOpenContainer` as the attributes argument.

```
CMValue  CMGetNextValue(CMObject object,
                        CMProperty property,
                        CMValue currValue)
```

This routine returns the refNum for the next value (according to the current value order) in the objects property following `currValue`. If currValue is `NULL`, the refNum for the first value for that object's property is returned. If `currValue` is not `NULL`, the next value for that object's property is returned. `NULL` is returned if there are no more type values following `currValue` or the object does not contain the property.

`currValue` is generally a refNum previously returned from this routine. Successive calls to this routine will thus yield all the values for the specified property of the specified object as long as no other operations change the value order.

```
CMValue CMNewValue(CMObject object,
                   CMProperty property,
                   CMType type,
                   ...);
```

A new entry is created for the designated object, with the designated property and type and a refnum to the entry is returned. The generation number of the value defaults to the generation number of the container, but it may be set with `CMSetValueGeneration`.

An object's properties can have more than one value. However, the all the types for the values belonging to a given object property must be **unique**. It is an error to attempt to create a value for a property when there is already a value of the same type for that property.

If the specified type corresponds to a global type name that has an associated "use value" handler, or if its base types (if any) have associated "use value" handlers, a dynamic value will be created and returned. The value will be initialized using the `dataInitParams` arguments, which must correspond to the initialization arguments for a value of that type. See Chapter 5: Types and Dynamic Values for details.

Note that the value refnum at this point has no associated data. The value data is set with `CMWriteValueData` or `CMOpenNewContainer` (to write an embedded container). If the value will be used as an embedded container it must have the embedded container type. Using `CMWriteValueData` on a value of this type is an error.

The value is created at an unspecified location in the sequence of values for the specified property. Creating a new value may cause the order of the values for that property to change.

```
CMValue   CMVNewValue(CMObject object,
                      CMProperty property,
                      CMType type,
                      va_list dataInitParams)
```

This routine is the same as `CMNewValue()` above, except that the dynamic value data initialization (i.e., "...") parameters are given as a variable argument list as defined by the "stdarg" facility.

This routine assumes the caller sets up and terminates the variable arg list using the "stdarg.h" calls as follows:

```
#include <stdarg.h>

callersRoutine(args, ...)

{

va_list dataInitParams;

- - -

va_start(dataInitParams, args);

value = CMVNewValue(object, property, type, dataInitParams);
```

```
va_end(dataInitParams);

- - -

}

CMSize CMGetValueSize(CMValue value);
```

The size of the designated value is returned.

If the storage size of the value is different from its size as seen by the application (for example, if it is compressed) the value handlers are responsible for keeping track of the application visible size and responding correctly. See Chapter 5: Types and Dynamic Values for details.

```
CMSize CMReadValueData(CMValue value,
                       CMPtr buffer,
                       CMCount offset,
                       CMSize maxSize)
```

The data, starting at the offset, for the value is read into the buffer. The size of the data read is returned. Up to `maxSize` characters will be read (can be 0).

The data is read starting at the offset, up to the end of the data, or maxSize characters, whichever comes first. Offsets are relative to 0. If the starting offset is greater than or equal to the current data size, no data is read and 0 returned.

It is an error to attempt to read a value which has no data, i.e., a value where only a `CMNewValue` has been done.

```
void CMWriteValueData(CMValue value,
                      CMPtr buffer,
                      CMCount offset,
                      CMSize size)
```

The buffer is written to the container and defined as the data for the value. If the value already has data associated with it, the buffer overwrites the "old" data starting at the offset character position. `size` bytes are written.

If the current size of the value data is T (it will be 0 for a new value created by `CMNewValue`), then the offset may be any value from 0 to T+1. That is, existing data may be overwritten or the value extended with new data. The value of T can be gotten using `CMGetValueSize`. Note that no "holes" can be created. It is an error to use an offset greater than T+1.

Once data has been written to the container, it may be read using `CMReadValue-Data`. `CMWriteValueData` may only be used on an updateable container.

`CMWriteValueData` calls for a particular value do not have to be contiguous. Writes for other values can be done in between writes to a given value. The library takes care of generating separate value segments. The data is physically not contiguous in the container in this case. `CMWriteValueData` and `CMReadValueData` hide this by allowing the user to view the data as contiguous. The input offset is mapped to the proper starting segment and to the offset within that segment.

`CMWriteValueData()` may only be used for a container opend for writing (or converting) using `CMOpenNewContainer()`. It is an error to write to protected values, which are created implicitly by the API. This includes the predefined TOC objects (seed and offset values) and objects representing currently opened embedded containers.

For creating embedded containers, `CMOPenNewContainer` is used instead of `CMWriteValueData`. See `CMNewValue` and `CMOpenContainer` for further details.

```
VOID CMInsertValueData(CMValue value,
                       CMPtr buffer,
                       CMCount offset,
                       CMSize size)
```

If the current size of the value data is T, `offset` must be ≤ T+1. The existing data in the value is "pushed aside" and the buffer is written in the space created.

```
                           0              offset              offset+size  T+size
       Existing value data:
             Inserted data:
                     Size:
                   Offset:
```
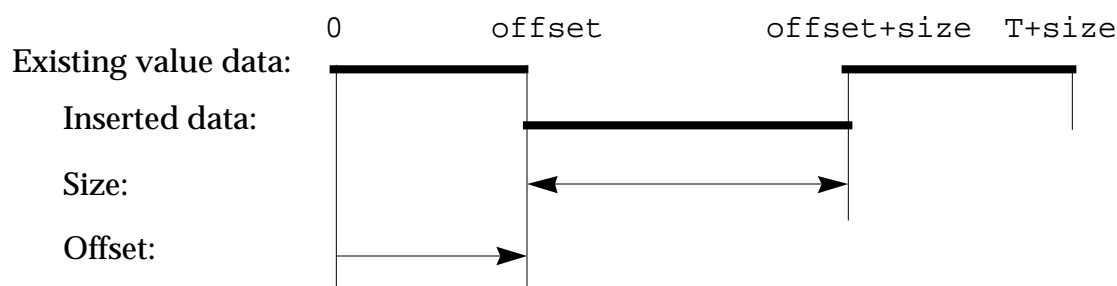
```
VOID CMDeleteValueData(CMValue value,
                       CMCount offset,
                       CMSize size)
```

Let T be the length of the value data. The bytes from `offset` to `offset + size` are deleted from the value, and the value is "closed up". After this operation, the size of the value data is T-`size` (assuming `offset`+`size` is ≤ T). If `offset` is greater than T, no data is deleted. If `offset`+`size` is greater than T, all the data from `offset` to T is deleted. Neither case produces an error.

```
                           0              offset              offset+size  T-size
       Existing value data:
              Deleted data:
                     Size:
                    Offset:
```
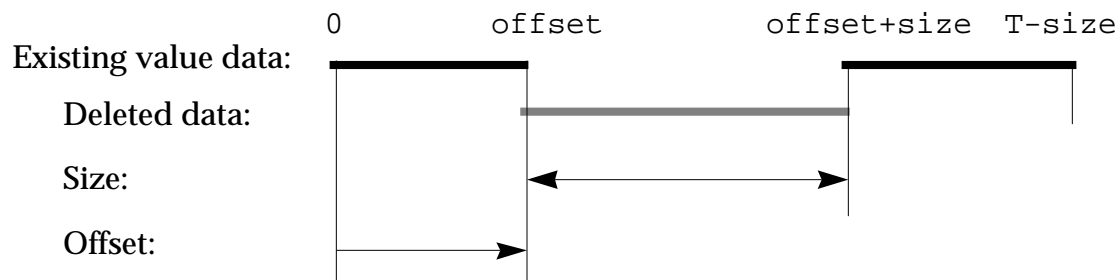
```
VOID CMDefineValueData(CMValue value,
                       CMSize offset,
                       CMSize size);
```

Existing data in the container, which must have been in the container when it was opened by Bento, is defined as the data for the value. No data is written to the container. The container must have been opened using `CMOpenNewContainer`, with the flag `kCMConverting` set in the `useMode`.

The designated value is set to reference the indicated data. The offset given is the offset from the beginning of the container. It is an error to give an offset or a size that would result in the value containing bytes outside of the data that was in the container when it was opened. The offset therefore, must be in the range of 0 to N-1, where N is the size of preexisting data at the time the container was opened.

Additional calls to CMDefineValueData() for the **same** value will define additional. i.e., continued, segments when the offset produces noncontiguous data definition. If the size of the last (most recent) value segment is T, and the offset for that segment is such that offset+T equals the offset for the additional segment, then the last segment is simply extended. This follows the same rules as `CMWriteValueData()`.

```
void  CMMoveValue(CMValue value,
                  CMObject object,
                  CMProperty property)
```

Moves the specified value from its original object property to the specified object property. The value is physically deleted from its original object/property as if a `CMDeleteValue()` were done on it. If the value deleted is the only one for the property, the property itself is deleted as in `CMDeleteObjectProperty()`.

The value is added to the "to"s object propery in the same manner as a `CMNewValue()`. The order of the values for both the value's original object property and for the value's new object property may be changed.

Note, that although the effect of a move is logically a combination `CMDeleteValue()` and `CMNewValue()`, the refnum of the value remains valid. Its association is now with the new object property.

This operation may be done at any time. No data need be assoicated with the value at the time of the move. Only moves **within the same container** are allowed.

```
VOID CMGetValueInfo(CMValue value,
                    CMContainer *container,
                    CMObject *object,
                    CMProperty *property,
                    CMGeneration *generation);
```

The container, object, property, and generation of the designated entry are returned. `NULL` may be passed for any argument except the first.

```
void  CMSetValueType(CMValue value,
                     CMType type)
```

The type of the value is set as specified.

```
void  CMSetValueGeneration(CMValue value,
                           CMGeneration generation)
```

The generation for the specified value is set. The generation number must be greater than or equal to 1. Normally this routine doesn't need to be used since the value inherits its generation from its container.

```
void CMDeleteValue(CMValue value);
```

The designated value is deleted from its object property. A deleted value will be treated by all Bento operations as though it does not exist. For example, it will not be found by CMUseValue, counted by CMCountValues, etc. .

If the value deleted is the only one for the property, the property itself is deleted as in CMDeleteObjectProperty. If that property is the only one for the object, the object is also deleted as in CMDeleteObject. Some values are protected from deletion. Protected values include the predefined TOC object values (seed and offset) and any currently open embedded container values.

```
void CMReleaseValue(CMValue value);
```

The association between the Value refnum and the entry is destroyed. After this call the refnum is invalid, and may be returned from a subsequent CMUseValue or CMNewValue call to designate another value.

Reference Operations

```
CMReference    *CMNewReference(CMValue value,
                      CMObject referencedObject,
                      CMReference theReferenceData)
```

This is the only way to get a persistent reference to an object that can be saved in a value, and then read from the value and used to refer to that object when the container is opened in another environment. CMNewReference does some bookkeeping behind the scenes and returns a token (theReferenceData) that will refer to referencedObject, but this reference will only be valid in the context of value!

The caller should write the theReferenceData to the value. It can be embedded in any way in the value: encrypted, compressed, whatever you want. The size of theReferenceData is determined by the size of the CMReference type.

```
CMObject  CMGetReferencedObject(CMValue value,
                      CMReference theReferenceData)
```

Provides the object refNum corresponding to theReferenceData.

`value` must be the value that contained `theReferenceData`. Values from many containers may be present at the same time, and the caller may not be aware of what container a given reference is from, especially in the presence of I/O redirection. Furthermore, the reference may have been "fixed up" using the other references routines below. Such fixups only apply to a hidden reference table associated with the value, so the value must be used as the context for converting the persistent reference.

```
CMReference  *CMSetReference(CMValue value,
                 CMObject referencedObject,
                 CMReference theReferenceData);
```

This call is similar to `CMNewReference()` except that here the caller defines the `CMReference` key to associate with an object. The specified key must be a nonzero value. The (input) pointer to `theReferenceData` key is returned.

In all cases the specified `CMReference` key is associated with the specified `referencedObject`. These associations are maintained in a persistent table attached to the value. If `theReferenceData` key is new, a new reference is recorded. If `theReferenceData` key matches one of the previously recorded keys in the table the reference associated with that key is **changed** to associate it with the (new) `referencedObject`.

This call can be used to "fix up" existing references if a value is copied as part of a structure or moved to a new environment. It can also be used to associate object references with pre-existing keys in the value data.

The only difference between `CMNewReference()` and `CMSetReference()` is that with `CMNewReference()`, the Container Manager defines the `CMReference` key, while with `CMSetReference()` the caller can define the key. The net result is the same; the keys are recorded in a persistent form attached to the value.

Note that multiple references to the same object can be recorded by passing different keys as `theReferenceData`.

Once these associations are recorded, they may be counted, deleted, and accessed using `CMCountReferences()`, `CMDeleteReference()`, and `CMGetNextReference()` respectively.

```
void CMDeleteReference(CMValue value,
                  CMReference theReferenceData);
```

This call deletes a single object reference created by `CMNewReference()` or `CMSetReference()` associated with the `theReferenceData` key in the reference table attached to the value.

The value's reference table is searched for the specified `theReferenceData` key. If it is found, the association is removed. If it is not found this routine does nothing. It is not an error if the `theReferenceData` key is not found.

```
CMCount CMCountReferences(CMValue value);
```

Returns the total number of references in the reference table attached to the value. These references were recorded by `CMNewReference()` or `CMSetReference()`.

```
CMReference *CMGetNextReference(CMValue value,
                       CMReference currReferenceData);
```

This routine returns the next reference key following the `currReferenceData` key in the reference table for the specified value.

If `currReferenceData` is 0, then the first object reference key is returned. If `currReferenceData` is not 0, the next reference key after `currReferenceData` is returned. The next reference key is stored into `currReferenceData` and the pointer to `currReferenceData` is returned as the function result. `NULL` is returned and `currReferenceData` is undefined if there are no references following the key passed in as `currReferenceData`.

Chapter 5: Types and Dynamic Values

Bento provides a very powerful mechanism for transforming values during I/O, and for following indirect references. This chapter describes the way that types can be built to define such values, and explains how the library supports such types.

This entire chapter is new, so no change bars are used.

Usage Examples

The Bento type mechansims are probably best explained in terms of some usage examples.

External File

Suppose we would like to have a value that represents a file. When we do `CMWriteValueData` to the value, we want to actually perform I/O to the file.

The mechanism described in this chapter allows us to store a reference to the file in a value. When the value is `Used`, an I/O redirection is set up, without the application's being aware of it.

Note that this raises the thorny problem of platform-independent file references. Bento avoids this problem. It allows any number of different types of references, implemented by handers. Naturally, we intend to encourage definition of a standard platform independent file reference mechanism, but this is not required to use Bento.

Compressed Value

Suppose we would like to compress data as it is written to the value, and decompress it as it is read out. In addition to maintaining the data in the value itself, this compression may depend on a dictionary associated with the type of value. Furthermore, the compression routine may need to keep various state around, since the compression at any point may depend on what has already been written.

The mechanism described in this chapter allows us to give the value a type that causes the compression/decompression handler to be transparently invoked when the application does I/O. Again, this is an extensible mechanism, so that new compression algorithms (or more generally, arbitrary transformations) can be added without modifying the library.

Compressed, Format Converted Array

Suppose the value we are dealing with is actually an array of pixels. In addition to decompressing it, on a given platform we want to convert each pixel to a different format.

The mechanism described in this chapter allows us to take two (or more) data transformations, such as compression and format conversion, and compose them together. Just as the application does not need to be aware of the underlying transformations, the individual transformations do not need to be aware of each other.

<u>All of the Above</u>

Obviously, the next step is to put the compressed pixel array out in a file, and convert it to a different format when it is read in. This is all supported using exactly the same composition as used in the previous example. The interfaces to data transformations and I/O redirection are the same, so no special mechanism is required.

<u>Stranger (But Still Useful) Examples</u>

To briefly illustrate further where this leads, here are some more unusual examples:

• A value contains a query that is used to look information up in a database. The "I/O redirection" provides access to a table retrieved from the database.

• A value contains a file reference that is encrypted because it also holds the file-server password. A decryption stage is required before the I/O redirector can be applied to the file reference.

• A value contains a query that is used to generate a file reference, which then becomes the basis for a second level of I/O redirection.

But you get the idea.

Structured Types _____

All of these examples are based on the types of the values involved. The examples depend on two aspects of Bento types.

First, every value handler is bound indirectly through the name of a type. Handlers are associated with type names through the `CMSetMetaHandler` operation. This association is session-wide. Then the handler is bound to a particular type in a given container through the name of that type. This binding is done when the container is opened.

Second, even in the simplest examples above, such as the value that is just an indirection to a file, or the the value that is just compressed, the value essentially has two types: the type visible to the application, which encodes the format of the data from the application's point of view, and the type used to find the appropriate handler for compression, I/O redirection, etc.

As the more complex examples show, these multiple types of a value need to be independent. This leads to a view of a value as having multiple, independent types. By analogy with C++ (an analogy we will explore in detail below) we call these "base types" of the value type. Base types can be added to and removed from any Bento type using the `CMAddBaseType()` and `CMRemoveBaseType()` operations documented in the API chapter.

Base types are normal types, and themselves may have base types. This could be useful, for example, when the combination of file access and decompression is used in a variety of different contexts. The two could be made base types of a new type, and then that new type could be used in various ways, including making it a base type of the "all of the above" type which adds format conversion. A picture may clarify this somewhat:

```
File        Compres-        File        Compres-
Access      sion            Acces       sion
     ◯          ◯                ◯          ◯  Convert     Final
                                                 Format      Type
          ◯                        Comp.  ◯       ◯          ◯
       Comp.                       File
        File                                   ◯
                                          All of the
                                          Above
```
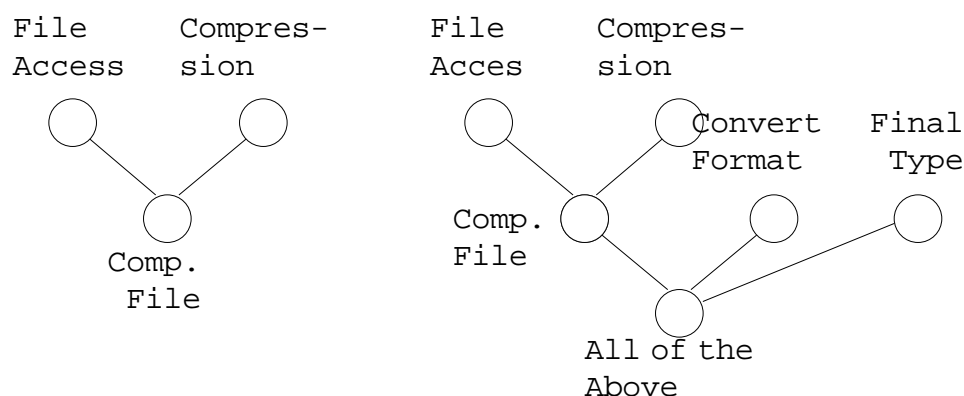
Base types will always form a tree rooted in the original type. If the same type is used as a base type in more than one place in the tree, the separate uses are treated as entirely separate types.

To understand how a given set of base types will behave, we must flatten them out into a linear string. This is done by performing a depth-first, post-order walk on the tree. Thus, in the case above, the final order is File Access, Compression, Comp. File, Convert Format, Final Type, All of the Above. Because Comp. File and All of the Above do not have handlers (let us assume) they will not have any effect on the value.

Dynamic Values _____

Now that we understand in a general way what Bento does with types, we need to look more closely at how this works. The remainder of this chapter is probably only of interest to people who want to know what it would take to write a value handler. You may understand some parts better if you read the appendix on handlers first.

The examples above indicate a number of constraints on our design. First of all, the application, and each handler, must always think that it is dealing with a "normal" value. Second, in several cases we saw that handlers might have a non-trivial amount of state to manage.

We address these constraints by giving each handler its own "private" value, called a **dynamic value**. Dynamic values are transient (ie. not persistent); they are created just to provide an environment for the handlers, and they are never written to the container, saved in the TOC, etc. However, they do have refnums and from the "outside" (ie. from any application code or handler code except the handler that "owns" them) they look exactly like normal values.

This implies that the normal value operations must be supported for each dynamic value. The operations are supported by the API passing each operation to the handler that "owns" the dynamic value. (Actually, a few operations are executed by the API; we will discuss those later.)

The following value operations can be supported by each value handler:

```
CMSize CMGetValueSize(CMValue value);

CMSize CMReadValueData(CMValue value,
                      CMPtr buffer,
                      CMCount offset,
                      CMSize maxSize);

void CMWriteValueData(CMValue value,
                      CMPtr buffer,
                      CMCount offset,
                      CMSize size);

void CMInsertValueData(CMValue value,
                      CMPtr buffer,
                      CMCount offset,
                      CMSize size);

void CMDeleteValueData(CMValue value,
                      CMCount offset,
                      CMSize size);

void CMGetValueInfo(CMValue value,
                      CMContainer *container,
                      CMObject *object,
                      CMProperty *property,
                      CMType *type,
                      CMGeneration *generation);

void CMSetValueType(CMValue value,
                      CMType type);

void CMSetValueGeneration(CMValue value,
                      CMGeneration generation);

void CMReleaseValue(CMValue);
```

When a dynamic value is spawned by `CMNewValue()` or `CMUseValue()`, the pointer to the top-most dynamic value header is returned as the refNum. Then, whenever the user passes a refnum to an API value routine, it checks to see if the refNum is a dynamic value. If it is, it initiates the call to the corresponding value handler. That may cause a search up the base value chain to look for the "inherited" value routine. In the limit, we end up using the original API value routine if no handler is supplied and we reach the "real" value in the chain. Thus the handler must be semantically identical to the corresponding API call.

These dynamic values only exist from creation during the `CMUseValue()` (discussed below) until until they are released by `CMReleaseValue()`. A dynamic value can have its own data, but this data is stored in the value's refCon rather than in the value data itself. Dynamic values do not have associated data in the normal sense.

Dynamic Value Creation

A dynamic value is created when a value is created by `CMNewValue()` or used by `CMUseValue()`, and the following two conditions occur:

1.   The type or any of its base types have associated metahandlers registered by `CMSet-MetaHandler()`.

2.   The metahandlers support a Use Value Handler, and in addition for `CMNewValue()`, a New Value Handler.

The New Value Handlers are used to save initialization data for the Use Value Handlers. The Use Value Handlers are called to set up and return a refCon. Another metahandler address is also returned. This is used to get the address of the value operation handlers corresponding to the standard API CM... value routines mentioned above.

When a `CMNewValue()` or `CMUseValue()` is almost done, a check is made on the value's type, and all of its base types (if any) to see if it has an associated registered metahandler. If it does it is called with a Use value operation type to see if a Use Value Handler exists for the type. If it does, we spawn the dynamic value.

The spawning is done by calling the Use Value Handler. The Use Value Handler is expected to set up a refCon to pass among the value handlers and a pointer to another metahandler. These are returned to `CMNewValue()` or `CMUseValue()` which does the actual creation of the dynamic value. The extensions are initialized, the metahandler pointer and refCon are saved. The pointer to the created dynamic value header is what `CMNewValue()` or `CMUseValue()` returns to the user as the refNum.

Now, when the user attempts to do a value operation using this refNum, we will use the corresponding handler routine in its place. The vector entries are set on first use of a value operation. If a handler for a particular operation is not defined for a value, its "base value" is used to get the "inherited" handler. This continues up the chain of base values, up to the original "real" value that spawned the base values from the `CMNewValue()` or `CMUseValue()`. Once found, we save the handler in the top layer vector (associated with the refNum) so we don't have to do the search again. Thus, as in C++, dynamic values may be "subclassed" via their (base) types.

Note that if we indeed do have to search up the base value chain then we must save the dynamic value refNum (pointer) along with the handler address. This is very much like C++ classes, where inherited methods are called and the appropriate "this" must also be passed.

Layering Dynamic Values

The best way to describe layering is in terms of C++. Say we have the following class types (using a somewhat abbreviated notation):

```
class Layer1 {// a base class
<layer1 data>// possible data (fields)
Layer1(<layer1 args>);// constructor to init the data
other methods...// value operations in our case
 };


class Layer2 {// another base class
<layer2 data>// possible data (fields)
Layer2(<layer2 args>);// constructor to init the data
other methods...// value operations in our case
};


class T: Layer1, Layer2 {// the class of interest!
<T data>// possible data (fields)
T(<T args>, <layer1 args>, <layer2 args>);

                                    // constructor to init the data
                   and bases
other methods...// value operations in our case
};
```

In Container Manager terminology, T is to be a registered type with other registered types as base types (classes). All type objects are created using the standard API call `CMRegisterType()`. Base types can be added to a type by using `CMAddBaseType()`. This defines a form of inheritance like the C++ classes.

Type T would be registered with its base types as follows:

```
layer1 = CMRegisterType(container, "Layer1");

layer2 = CMRegisterType(container, "Layer2");


t = CMRegisterType(container, "T");

CMAddBaseType(t, layer1);

CMAddBaseType(t, layer2);
```

For the t object, the global name property and value are created as usual by `CMRegisterType(container, "T")`. The `CMAddBaseType()` calls add the base types. These are recorded as the object ID's for each base type in the order created as separate value segments for a special "base type" property belonging to the type object.

As mentioned above, `CMNewValue()` or `CMUseValue()` spawn dynamic values if the original type or any of its base types have an associated Use Value Handler. Assume that was done for "T" in the above example. What happens is that `CMNewValue()` or `CMUseValue()` will look at its type object (t here) to see if the base type property is present. If it is, it will follow each type "down" to leaf types using a depth-first search.

In the example, "layer1" will be visited, then "layer2", and finally the original type "T" itself. If the "layer1" type object had base types of its own, they would be visited before using "layer1" itself. Hence the depth-first search down to the leaf types.

For each type processed, if it has a Use Value Handler of its own, it will be called to get a refCon and value handler metahandler.

Note that this scheme allows total freedom for the user to mix types. For example, type T1 could have base types T2 and T3. Alternatively, T1 could just have base type T2 and T2 have T3 as its base type!

Data For Dynamic Values

In the C++ class types shown above, note that each class could have its own data along with its own constructor. The T class has a constructor that calls the constructors of all of its base classes. We can carry this analogy with the Container Manager just so far! Here is where it starts to break down.

The problem here is that C++ class types are declared statically. A C++ compiler can see all the base classes and can tell what data gets inherited and who goes with what class. In the Container Manager, all "classes" (i.e., our type objects) are created dynamically! So the problem is we need some way to tell what data "belongs" to what type.

The solution is yet another special handler, which returns a format specification-called metadata. The handler is the Metadata Handler whose address is determined by the Container Manager from the same metahandler that returns the New Value and Use Value Handler addresses.

Metadata is very similar to C `printf()` format descriptions, and is used for similar purposes. The next section will describe the metadata in detail. For now, it is sufficient to know that it tells `CMNewValue()` how to interpret its `"..."` parameters. The rest of this section will discuss how this is done to dynamically create data.

As with C++ classes, the data is created when a new value is created, i.e., with a `CMNewValue()` call. The data will be saved in the container, so `CMUseValue()` uses the type format descriptions to extract the data for each dynamic value layer.

`CMNewValue()` is defined as follows:

```
CMValue CMNewValue(CMObject object,
                   CMProperty property,
                   CMType type, ...);
```

The `"..."` is an arbitrary number of parameters used to create the data. Metadata, accessed from the Metadata Handler, tells `CMNewValue()` how to interpret the parameters just like a printf() format tells it how to use its arguments.

The **order** of the parameters is important! Because base types are done with a depth-first search through the types down to their leaves, the `CMNewValue()` `"..."` parameters **must** be ordered with the parameters for the first type in the chain occuring first in the parameter list. Note what's happening here is you are supplying all the constructor data just like T constructor class example above.

The way the data gets written is with a special handler, called the New Value Handler. After `CMNewValue()` calls the Metadata Handler, it uses the metadata to extract the next set of `CMNewValue()` `"..."` parameters. `CMNewValue()` then passes the parameters along in the form of a data packet to the New Value Handler. The New Value Handler is then expected to use this data, which it can extract with `CMScanData-Packet()` (see the handler appendix). Once it has the data, it can compute initialization values to **write** to its **base** value. It is the data written by the New Value Handler that the Use Value Handler will read to create its refCon.

Only `CMNewValue()` does this. The New Value Handler is only for new values, but the Use Value Handler is used by both `CMNewValue()` and `CMUseValue()`.

In the simplest case, with only one dynamic value, you can see that the data is written to the "real" value. Now if you layer another dynamic value on to this, the next chunk of data is written using that layer's base value and hence its handlers. The second layer will thus use the first layer's handlers. That may or may not end up writing to the "real" value depending on the kind of layer it is. If it's some sort of I/O redirection handler (i.e., it reads and writes somewhere else), the second layer data will probably not go to the "real" value.

The Use Value Handler is called both for `CMNewValue()` and `CMUseValue()`. The Use Value Handler reads the data from its base value to create its refCon. If the user comes back the next day and does a `CMUseValue()`, only the Use Value Handler is called. Again it reads the data from its base value to construct the refCon and we're back as we were before in the `CMNewValue()` case.

Handler Contracts

It should be pointed out here that the Metadata and New Value Handlers will always be executed with a Container Manager running on some particular hardware (obviously). The data packet built from the `CMNewValue()` `"..."` parameters is stored as a function of the hardware implementation on which it is run (i.e., whatever the sizes are for bytes, words, longs, etc.). How it is stored is a function of the metadata returned from the Metadata Handler. In other terms, the New Value Handler has a contract with both the Container Manager and the Metadata Handler on the meaning of the parameter data.

Note, however, it is **not** required that you be on the same hardware when you come back the next day and to the `CMUseValue()` that leads to the Use Value Handler call. The handler writer must keep this in mind. Specifically, the Use Value Handler **must** know the attributes (bytes size, big/little endian, etc.) of the data written out by the New Value Handler so it knows how to use that info. In other words, the Use Value Handler has a (separate) "contract" with its own New Value Handler on the meaning of the data written to the base value.

There is another, relatively minor, thing to keep in mind. That is that the value handlers for any one layer must take into account the size of its own data when manipulating additional data created by the handlers for `CMReadValueData()`, `CMWriteValueData()`, etc. This simply offsets the write and read value data operations by the proper amount. Remember all operations are on their base values. So if a New Value Handler writes data, this basically prefixes the "real" stuff being written by the handler operations.

Metadata

As mentioned above, the metadata directs CMNewValue() on how to interpret its "..." parameters to build data packets passed to New Value Handlers.

The format string is a sequence of characters containing data format specifications. Unlike `printf()`, anything other than the data format specifications are ignored. They are assumed to be comments.

The data format specifications indicate to `CMNewValue()` how to interpret its data initialization parameters. Each specification uses the next corresponding "..." parameter to `CMNewValue()`. This is similar to the behavior of `printf()`.

A data format specification begins with a "%" sign. Immediately following the % is a required data format descriptor, expressed as a sequence of characters. The data format descriptors are as follows (numbers in "[]" indicate notes following the format descriptor list):

c   A character or byte [1].

d   A short [1].

l[d]A long (the "d" is optional) [1].

[*]sA C string (i.e., null delimited). Optionally a "*" indicates that only the first n characters of the string are to be used. The "*" consumes an additional `CMNewValue()` "..." parameter of type `CMSize` [3, 4, 5].

i   An object, property, or type ID. Thus is defined as the same size as "ld" [1].

p   A pointer [1, 2].

o   A `CMObject`, `CMProperty`, or `CMType` object refNum. This is defined as the same size as "p" [1, 2].

v   A `CMValue` refNum. This is defined as the same size as "p" [1, 2].

Notes:

1.   The `CMNewValue()` "..." parameters are converted to a packet of data using the hardware implementation defined sizes for bytes, words, longs, etc. as directed by the metadata returned from the Metadata Handler. Thus the Metadata Handler has a contract with the New Value Handler that `CMNewValue()` calls. The data that the New Value Handler writes to its base value is in terms of a "contract" it has with its Use Value Handler. It is the one that will read that base value data to create its refCon. If `CMUseValue()` is expected to be run on different hardware with different byte sizes, endianess, etc, then that is between the New Value Handler and its Use Value Handler. The Container Manager is independent of that.

2.   Pointers can be passed to `CMNewValue()` to convey special information to the Use Value Handler. You shouldn't, of course, write these as data. RefNums can be passed to extract object ID's or other read value data. It is permissable to write object ID's to data. But this will put a restriction on such referenced objects that they shouldn't be moved or deleted.

3.   For "`%*s`", the value corresponding to the "`*`" is copied to the packet data immediately in front of the string. This is somewhat (not quite) equivalent to "`%l%s`", where the `%l` is the length, n, and `%s` is a n byte string. Note however, this string is **not** null delimited.

4.   Caution, the string will be **copied** from the string pointed to in the `CMNewValue()` "`...`" parameter list to the packet. It you intend to pass a pointer to the string, rather than the string itself, `%p` should be used. Frankly, `%s` will not be used much.

5.   For symmetry `CMScanDataPacket()` returns the value of string length to an explict distinct parameter pointer. Thus the parameter pointer list passed to `CMScanData-Packet()` should be identical to the "`...`" parameters passed to a `CMNewValue()` "`...`" parameter list (at least the portion corresponding to this type).

The Metadata, New Value, and Use Value Handlers

The Metadata Handler is only needed for `CMNewValue()` so that the proper number of `CMNewValue()` "`...`" parameters can be placed into a data packet for the New Value Handler.

The Metadata Handler must have the following prototype:

```
CMMetaData metaData_Handler(CMType type);
```

where

`type` = the (base) type layer whose metadata is to be defined.

The Metadata Handler simply returns a C string containing the metadata using the format descriptions described above.

The type is passed as a convenience. It may or may not be needed. It is possible for a type object to contain **other** data for other properties. Types, after all, are ordinary objects. There is nothing prohibiting the creation of additional properties and their values. This fact could be used to add additional (static and private) information to a type to be used elsewhere. For example, the type could contain a compression dictionary.

Note, as in `printf()`, if the metadata handlers "lie" about the metadata format, or if there aren't enough parameters supplied to `CMNewValue()`, the results will be unpredictable!

The New Value Handler must have the following prototype:

```
CMBoolean newValue_Handler(CMValue baseValue,
                   CMType type,
                   CMDataPacket dataPacket);
```

where

`baseValue` = the base value which is to be used to write the refCon data for the Use Value Handler.

`type` = the type corresponding to this New Value Handler.

`dataPacket` = the pointer to the data packet, created from the `CMNewValue()` "..." parameters according the types metadata format description.

The type is passed again as a convenience just as in the Metadata Handler. It can also be used here to pass to `CMScanDataPacket()` to extract the dataPacket back into variables that exactly correspond to that portion of the `CMNewValue()` "..." parameters that correspond to the type. It is not required, however that `CMScanDataPacket()` be used.

The Use Value Handler is called for both the `CMUseValue()` and `CMNewValue()` cases. If its companion New Value Handler wrote data to its base value, the Use Value Handler will probably read the data to create its refCon. The refCon will be passed to all value handlers. The Use Value Handler returns its refCon along with another metahandler address that is used to get the value handler addresses. These are used to create the dynamic value.

The Use Value Handler should have the following prototype:

```
CMBoolean useValue_Handler(CMValue baseValue,
                    CMType type,
                    CMMetaHandler *metahandler,
                    CMRefCon *refCon);
```

where

`baseValue` = the base value which is to be used to write the refCon data for the Use Value Handler.

`type` = the type corresponding to this New Value Handler.

`metahandler` = a pointer to the value operations metahandler which is **returned** by the Use Value Handler to its caller.

`refCon` = a reference constant built by the Use Value Handler and **returned** to its caller.

The baseValue and type are identical to the ones passed to the New Value Handler. The type may or may not be needed in the Use Value Handler. Like the Use Value Handler, it could be used to supply additional information from other properties.

It is expected that the Use Value Handler will read data from its base value to construct its refCon. The refCon is then returned along with a pointer to another metahandler that is used by the Container Manager to get the addresses of the value operations.

Note, both the New Value and Use Value Handlers return a `CMBoolean` to indicate success or failure. Failure means (or it is assumed) that the handlers reported some kind of error condition or failure. As documented, error reporters are not supposed to return. But in case they do, we use the `CMBoolean` to know what happened. It should return 0 to indicate failure and nonzero for success.

Value Operation Handlers _____

The value operation routines can do a `CMGetValueRefCon()` on the value passed to get at the refCon set up by the Use Value Handler. This provides a communication path among the value handlers. Further, the value handler should usually do its operations in terms of their base value, which can be accessed using CMGetBaseValue().

There is one exception to this rule; the release handler. A set of one or more dynamic value layers are spawned as a result of a single `CMUseValue()` or `CMNewValue()`. The layers result from the specified type haveing base types. From the caller's point of view s/he is doing one `CMUseValue()` or `CMNewValue()` with no consideration of the base types. That implies that the returned dynamic value should have a single `CMRelaseValue()` done on it. The handlers have no business doing `CMReleaseValue()` on their base value. This is detected and treated as an error.

A count is kept by the Container Manager of every `CMUseValue()` and `CMNewValue()`. Calling `CMReleaseValue()` reduces this count by one. When the last release is done on the dynamic value (its count goes to 0), the release handler will be called. It is the Container Manager who calls the release handler for all the layers, not the handler. The Container Manager created them as a result of the original type; it is therefore responsible for releasing them.

The reason the Container Manager is so insistent on forcing a release for every use of a dynamic value is mainly to enforce cleanup. Most value operation handlers will, at a minimum, use a refCon that was memory allocated by the Use Value Handler. Release handlers are responsible for freeing that memory. In another example, if any files were open by the Use Value Handler, the releases would close those files.

If all a value operation does is get its base value and call back the API routine to do its operation (again except for the release handler), then what it is basically doing is invoking the "inherited" value operation. In this case, the value operation could be **omitted** entirely by having the metahandler return NULL when asked for that operation. The Container Manager uses that as the signal to search up the dynamic value inheritance chain to find the first metahandler that **does** define the operation. In the limit, it will end up using the original "real" value.

Possible Limitations On Value Operations

Value I/O operations are basically stream operations. That is, you read or write a chunk of stuff linearly from a specified offset. In addition, Bento provides insert and delete operations (`CMInsertValueData()` and `CMDeleteValueData()`).

Insert and delete can cause problems because base types may want to do certain transformations on their data that depend on what has occurred previously in that stream of data. For example, encryption using a cyclic key, or compression generally cannot be done simply by looking at a chunk of data starting at some random offset. A cyclic key encryption can be deterministic if you can always determine where to start in the key as a function of offset. But you can see that inserts and deletes will change the offsets of following data. You would not know where to start in the key.

What all this means is that certain data transformations only make sense if you are willing to refuse to support the insert/delete operations. Basically only data transformations that are position independent can be supported with the full set of value operations.

Even simple I/O to a file may create problems, since most file systems do not support inserts and deletes in the middle of a file. If you do want to support inserts and deletes, then you should consider the potential for data intensive and/or computationally intensive operations.

Chapter 6: Format Overview

This conceptual description of the Bento format is intended as a road map to the detailed specification. As such, it says what but not how. When we get to the detailed level, we will see various caveats and tricks which are passed over in silence at this level. However, the format is very simple, so the details actually concern specific usage conventions more than extensions to the basic structure.

Key Ideas _____

There are four key ideas in the Bento format:

1)   everything in the container is an object,

2)   objects have persistent IDs,

3)   all the metadata lives in the TOC (Table of Contents),

4)   objects consist entirely of values, and

5)   each value knows its own property, type, and data location.

Let us discuss these in turn.

Everything is an object

In an Bento container, every accessible byte is part of a value of some object. Even the metadata that defines the structure of the container, and the label of the container, are values of an object. Type descriptions are objects, property descriptions are objects, etc. We will exploit this fact in various ways below.

Objects have persistent IDs

Every Bento object is designated by a persistent ID which is unique within the scope of its container. Objects may have additional IDs and/or names that are unique in larger scopes, but this is not required.

Object IDs provide a compact, convenient way to refer to an object. Clearly, we must provide an efficient mechanism to get from any object ID to information about that object.

All the metadata lives in the TOC

This is a difference between Bento and most other container formats, such as ASN.1, formats derived from IFF, etc. In these other formats, the metadata is associated with the chunks of data that it describes, a design approach that we call **internally tagged**. Jerry Morrison, the designer of IFF, was one of the major influences on the design of Bento; it was partly his experience with IFF that led us to move away from internal tagging.

There are three reasons for this difference from other formats:

a)   Bento needs to support very flexible layout, such as multi-media interleaving, and internal tags would be inconvenient and even harmful for this.

b)   Applications inspecting an object can make decisions about it more efficiently if all of its metadata is concentrated in one place, rather than being spread out over the container with its values.

c)   We want to be able to assimilate existing formats that contain collections of objects without forcing them to change. This implies that we must be able to designate regions within the existing structure as values, without forcing them to somehow retrofit internal tags.

Note that putting all the metadata in the TOC does not increase the amount of storage required. If we have a TOC for random access, it has to contain an object ID and an offset to each value in any case. Moving the type and size of the value to the TOC does not use more storage; it even uses less if the object ID also would have been stored in the object header.

This approach to metadata does impose one significant design constraint. A Bento container can only be read by starting with the TOC. This raises two questions: (1) how do we find the TOC, and (2) how do we access the TOC when we need information?

1)   In standard Bento containers the container label points to the TOC.

Possibly some non-standard containers will exist that require other mechanisms (discussed below). However, these will be exotic cases.

2)   Since we need to access the information in the TOC whenever we want to read a value, we have to have it available at all times. This normally means that the container needs to be on a random access device.

If a container needs to be read on a device that does not support efficient random access (such as a CD-ROM) the TOC can be split up into sub-TOCs that sit in front of the groups of objects they describe, and then the container can be accessed largely in stream order.

Objects consist entirely of values

In Bento, an object has no value as such. Each object has properties, and each property has values. The Bento format provides no information about an object except its ID.

Of course, an object can have a single value; in that case the value of the property "is" the value of the object. Thus we can easily accomodate the "normal" case.

Each value knows its own property, type, and data location

Each value consists of a property ID (or role), a type (or format), and data. For example, a graphic object might have a value that describes its "clip mask"; the property ID would specify what role the value plays, but not what format it is stored in. The type would define how the mask is represented: rectangle, bit mask, path, Mac region, PostScript path, etc. The data would be the representation of the mask itself.

At the level of the container standard itself, there are no restrictions on what values an object can have, how many values it can have, etc. However, individual object formats may dictate rules in this area. In general, applications should be prepared to encounter additional values that they do not understand; these can be ignored. This allows other applications to annotate objects with additional values that may not be generally understood. Typically, these values will be associated with properties that are unknown to the application.

The data of a value is an uninterpreted sequence of bytes which may be from 0 to 2^{64} bytes long. (The current Bento library only supports access to 2^{32} bytes per value, but the format is defined to support up to 2^{64}.) This sequence of bytes has no format requirements or restrictions. Furthermore, the byte sequences representing the data for various values of various objects can be placed anywhere in the container. Thus there are no strong data format requirements for the container as a whole, although it must contain the metadata to define its structure somewhere.

Special Cases

All of the mechanisms above are consistent across all the uses of objects. However, there are two special cases that need to be considered.

Multiple values

The format allows a single object to have multiple values with the same property ID. All the values must have different types. Such multiple values are intended to be used as alternative representations of the same information.

Value segments

The table of contents can contain multiple entries for a single value. These entries mean that the value represented by the entry consists of multiple segments.

This permits values to be broken up into chunks and interleaved, without creating problems for applications that view them as single values. In addition, it allows an application to build TOC entries that "synthesize" a value out of separate parts, as is required in retrofitting some file formats.

Note that these two special cases can be mixed freely. A property can have multiple values, and one or more of the values can be composed of multiple segments.

Other Issues

Globally Unique Names

To fulfill the requirement for locally generated unique names for types and properties, we have chosen to use the identifiers defined in ISO 9070. These are names that begin with a naming authority (assigned to a system vendor or an application vendor), and then continue with a series of more and more specific segments, until they end in a specific type or property name.

Names generated in this way are both unique and self-documenting. Individual users can generate unique names using this approach. For example, a user developing educational stackware might want to create properties, or even types, to use in scripts. The stackware development environment could automatically generate a unique prefix for the user, based on the serial number of the development tool, and then append the user generated property or type name.

This ensures that if that user's scripts and data are combined in a container with other information generated by other users, no naming conflicts can occur.

Note that globally unique names are not limited to property and type descriptions. Any object can be given a unique name using exactly the same mechanism, and such object names may be useful in some applications.

Note also that objects can be given short names that are only locally unique, as in the RIFF TOC. These would be a different type than Globally Unique Names.

Type and Property Descriptions

Recall that type and property descriptions are objects as well. What should we put in such descriptions?

Since types and properties need to have globally unique names, so that applications can recognize them, type and property descriptions will typically have a globally unique name value. In many cases, this may be the only contents of a description.

Sometimes, however, we may wish to put more information into a description. Here are some examples of useful information that can be attached to types or properties:

Base types

Base types allow us to inherit semantics from other existing types and compose it into more complex types. See the discussion of types and dynamic values for more details.

Note that the base type information is intended to include uses such as encryption, compression, I/O redirection, etc.

Encoding information

A type definition may indicate the default encoding of its values.

Typically, all of the values with the same basic format in a container will have the same encoding, so this new subtype can be shared by all these values. In this case the encoding can be indicated directly in the type description for the format.

If values with the same basic format but multiple encodings exist in the same container, a more complex solution is required. In this case, a subtype may be created just to record the encoding. Such a subtype will typically not need a globally unique name.

Compression information

In addition to the compression technique, typically recorded via a base type, the type can record compression parameters, the codebook used (if applicable), etc.

As with encoding information, a type that exists just to record compression information typically will not need a globally unique name. It will refer to the underlying format type and the compression technique, both of which will have globally unique names.

A template or grammar for a type

This allows applications that have never seen this type before to parse values of that type and potentially get some useful information out of them. Examples of description mechanisms that could be used in this way are ASN.1 and SGML.

The more general type will be indicated as the super-type. For example, a given SGML DTD as a type will have a specific SGML definition of the DTD. The super-type of this type would be SGML itself, which defines the basic encoding conventions.

Method descriptions for a type

A type could have properties that provide method definitions. Providing methods in the container would allow fully encapsulated use of values.

Consistency

Given these mechanisms, some objects may have a large and varied set of property values. Furthermore, containers may be copied with modifications by applications that do not understand all the property values of each object. Often we would like such applications to retain the values that they do not understand, but this naturally raises a major question of consistency. For example, if two values are alternative representations of the same information, and one of them is edited, but the editing application does not understand the other format, then the old value is out of date, and the object as a whole is inconsistent.

To give applications a handle for managing this problem, we define generation numbers. When a container is created for the first time, it has a generation number of 1. Each time a container is copied with modifications, its generation number is incremented. Each property entry contains the generation number of the last generation in which its value was modified (presumably by an application that understood its format). This allows an application to compare several values that should be consistent, and determine whether they were all modified together, and if not, which ones were modified most recently.

Example 1: A Bento container might hold a PICT and a GDI value representing the same drawing. These could be stored as values of the same drawing object. If this drawing is edited by an application that only understands GDI, the PICT value would be out of date, and an application that wanted a current PICT would need to invoke translation services to get a new version.

Example 2: A container might have extra indexes for the TOC, but these might not be understood by all applications that use the container, and it might be modified by an ignorant application that doesn't update its index. Comparison of the generation number of the TOC itself and the generation number of the index would immediately indicate whether they were in synch.

In addition, if desired, a container could retain information about the date and time each generation was created, and perhaps the individual responsible. This information could be provided as properties of the TOC itself.

Finding the Table of Contents

Obviously, given the importance of the table of contents to the structure of an Bento container, the first thing an application needs to do when accessing a container is find the table of contents. The standard mechanism for locating the TOC makes this easy.

The obvious mechanism would be to have a label at the beginning of the container (typically a file) that contains the offset of the table of contents. However, this turns out not to be quite the right approach.

In many cases, developers want to convert their existing file formats into Bento containers. Ideally, they would like to keep the resulting files readable by their existing applications that do not understand Bento. For example, we may be wrapping Bento around a RIFF file, and we may want the result to still be readable by an application that only understands RIFF. If the file has to have a standard Bento label at the beginning, it will be unreadable, unless the Bento label is specially designed to be compatible with RIFF. In that case, the label will be incompatible with some other format. Etc.

Our solution to this is to define the standard Bento format to have the label at the **end** of the container. This makes backward compatibility easy. In addition, a copy of the label can optionally be placed at the beginning of the container. This allows easy recognition of a Bento container if it is coming in from a stream-oriented I/O mechanism.

In exotic cases, this approach may not work. For example, transmission streams may depend on framing information, so that the stream can be read starting anywhere. Bento supports such exotic cases by allowing the TOC to be found in non-standard ways when necessary. Finding the TOC is the responsibility of the I/O handler, which will depend on the system and the container type. The I/O handler can adopt a non-standard approach if required.

Chapter 7: Format Definition

In this chapter, we will describe the concrete format of the file label and the table of contents. The motivation for the design choices is given in the previous chapter.

The format is effectively completely new, although it still encodes essentially the same information. Changes to this chapter are not indicated by change bars because almost everything would be marked.

The new format accomodates 64 bit value offsets, 32 bit generation numbers, and reference tables for values. At the same time, the format results in much lower overhead for most TOC entries.

Container Label Format_____

As mentioned in the previous chapter, a standard Bento container must have a label at the end. Exotic containers may have labels in other locations. However, the label will always provide the same information.

 The label contains the smallest possible amount of information, because it has to be the most stable part of the standard. It consists of seven fields:

Magic byte sequence

```
unsigned char magicBytes[8];
```

This identifies the container as a Bento container. It must be chosen to be unlikely to occur as the initial or final bytes in any existing file.

Label flags

```
unsigned short flags;
```

These are currently unused.

Buffer size

```
unsigned short blockSize;
```

This defines the size of the TOC blocks in this container, in multiples of 1024 bytes. This used to be the encoding field, which is no longer needed because the TOC now has a fixed encoding.

Container format major version number

```
unsigned short majorVersion;
```

The major format version number changes only on incompatible format changes.

Container format minor version number

```
unsigned short minorVersion;
```

The minor version number changes on upward compatible format changes. When the major version number changes, the minor version number is set to zero.

Offset of TOC

```
unsigned long tocOffset;
```

This provides the offset of the top level TOC (Table of Contents) from the beginning of the container. The TOC describes the structure of the rest of the container.

Size of TOC

```
unsigned long tocSize;
```

This provides the size of the TOC in bytes.

A picture of the Bento label format is as follows:

8 bytes	2 bytes	2 bytes	2 bytes	2 bytes	4 bytes	4 bytes
Magic Byte Seq.	Flags	blkSz	Maj.V	Min.V	TOC Off.	TOC Size

We do not expect additional information to be added to the TOC label. Instead, it can be added as values of the distinguished object #1, which represents the container itself.

Table Of Contents Format_____

Understanding the TOC is the key to understanding how the Bento format works. We begin with an overview, and follow up with the concrete details.

The TOC consists of a sequence of entries. Each entry corresponds to a single segment of a value of some object.

TOC entries are sorted by object ID, and within a single object they are sorted by property ID. This gives us two things:

•   All the entries for a given object are contiguous in the TOC, and all the entries for a given property are contiguous within the object.

•   We can find an object within the TOC, or a property within an object, by binary search. If an object ID or a property is not defined, we can quickly determine that it is not defined.

Thus, each object in the container is represented in the TOC by a sequence of entries, one for each segment of a value of the object.   (Bento has no way to represent an object without at least one value.)

Since each TOC entry defines a value, we know immediately that it must indicate the object ID, property, type, and data of the value. In addition, it indicates the generation number of the value and it may contain additional bookkeeping information for the value. Let us examine each of these in detail:

Object ID

The ID of the object that this value is part of.

Property

The property is indicated by the object ID of a property description.

Type

The type is indicated by the object ID of a type description.

Data

> The value is indicated by the offset and length of the sequence of bytes representing the value. The offset is a 0 origin byte offset from the beginning of the container. The length is a byte count, and may be 0, indicating a 0 length value.
>
> If the data is four bytes long or less, it may be included directly in the TOC as an **immediate value**, rather than being referenced by offset and length.

Generation number

> This allows applications to check consistency between different properties.

Bookkeeping information

> This field provides some additional information about the entry. It is described in more detail below.

Table Of Contents Low-Level Design

A TOC entry could simply be defined by putting all the information above in a record. Unfortunately, this record would be relatively large and would be very likely to contain redundant and/or unused information.

Specifically, the object ID, property, type, generation number, and bookkeeping information are each 4 bytes long. The data length is also four bytes, and the data offset can be as much as eight bytes. This is a total of 32 bytes, which would impose a significant overhead on small values.

Furthermore, sequential TOC entries are very likely to have the same object ID and generation number. They are may have the same property, and even the same type. (This last case means that the sequential entries are segments of the same value.) Many TOC entries will not need any bookkeeping information. Thus a scheme that requires all these fields for each TOC entry imposes considerable unnecessary overhead.

Instead, we have adopted an approach in which each TOC entry contains only the information that is new or different compared with the previous TOC entry. This results in a TOC that is organized as a stream rather than a table, and is parsed as it is read in. Due to the compression, we have found that reading in the new TOC format is significantly faster than reading in the old tabular format.

We will describe the TOC format at two levels: first, the logical structure of the stream, and second the actual physical representation.

Logical Stream Structure

The following grammar describes the set of stream elements and how they can be combined. Terminals are given in **bold**; they are described in detail in the list following the grammar. "*" means repeated 0 or more times. "+" means repeated 1 or more times. Square brackets mean present 0 or 1 time.

```
TOC                ::= object-ID property+

property           ::= property-ID value+
```

```
value                ::= type-ID [gen-num][ref-obj-ID]value-data

value-data           ::= initial-data continued-data*

initial-data         ::= immediate | short-reference
                         | long-reference

continued-data       ::= ctd-immediate | ctd-short-ref
                         ctd-long-ref
```

Object-ID, Property-ID, Type-ID, Ref-obj-ID

These are all object IDs. An object ID is a 32-bit persistent indentifier. The namespace for object IDs is local to the container. Object IDs are assigned sequentially as objects are allocated, and the last ID allocated in the container is maintain as a property of object # 1.

Gen-num

This is a 32-bit generation number of the value. It can be set by default from the container, or explicitly for a given value. Generation numbers are intended to be assigned sequentially to each consistent update of the container. The generation number of the TOC value in object # 1 is the default generation number of the container.

Immediate, Ctd-immediate

These are 0, 1, 2, 3 or 4 byte data values. They represent the actual data of the value. Except for the 0 byte case, they are all packaged in a 4 byte field. The ctd-immediate is exactly the same as the immediate, except that it is flagged to indicate that it is a value segment other than the first.

Short-ref, Ctd-short-ref

These are references to the data for a value segment using a 32-bit offset and a 32-bit length. The ctd-short-ref is exactly the same as the short-ref, except that it is flagged to indicate that it is a value segment other than the first.

Long-Ref, Ctd-long-ref

These are references to the data for a value segment using a 64-bit offset and a 32-bit length. The ctd-long-ref is exactly the same as the long-ref, except that it is flagged to indicate that it is a value segment other than the first.

Physical Stream Format

To make the TOC readable without starting from the beginning, it is broken up into fixed size blocks. The block size in any given container is specified in its label, and can range from 2^{10} bytes to 2^{26} bytes.

At the beginning of each block, there is assumed to be no previous information, and all information is written to the TOC. Thus a block can be read in and interpreted in isolation.

The stream of elements in the TOC is annotated with format information to make it possible to parse it. This annotation is done by placing one-byte codes at each point where the stream is ambiguous. Each distinct code is followed by a specific sequence of fields.

The specific byte codes used int the TOC, and associated sequence of fields for each code are given in the following list. The fields correspond to the grammar terminals described above. All fields are four bytes long, except for long data offsets, which are eight bytes long.

`NewObject`   1U

Fields: object ID, property ID, type ID

Meaning: Full specification of the identity of a value in the container. Found only at the beginning of an object or the beginning of a block.

`NewProperty`   2U

Fields: property ID, type ID

Meaning: Information required to begin a new property in an existing object.

`NewType`   3U

Fields: type ID

Meaning: Information required to begin a new value in an existing property.

`ExplicitGen`   4U

Fields: generation number

Meaning: Generation number of this value is different from preceeding value, or this is the first value in a block.

`Offset4Len4`   5U

Fields: offset (4 bytes), length

Meaning: Reference to value data; the first data segment for the value.

`ContdOffset4Len4`   6U

Fields: continued offset (4 bytes), length

Meaning: Reference to value data; **not** the first data segment for the value.

`Offset8Len4`   7U

Fields: offset (8 bytes), length

Meaning: Reference to value data; the first data segment for the value.

`ContdOffset8Len4`   8U

Fields: continued offset (8 bytes), length

Meaning: Reference to value data; **not** the first data segment for the value.

`Immediate0`   9U

Fields: none

`Immediate1`   10U

Fields: immediate data (1 byte) stored in a 4 byte field

Immediate2   11U

Fields: immediate data (2 bytes) stored in a 4 byte field

Immediate3   12U

Fields: immediate data (3 bytes) stored in a 4 byte field

Immediate4   13U

Fields: immediate data (4 bytes) stored in a 4 byte field

ContdImmediate4   14U

Fields: continued immediate data (4 bytes) stored in a 4 byte field

Meaning: Immediate data but **not** the first data segment for the value.

ReferenceListID   15U

Fields: object ID

Meaning: The ID of a bookkeeping object associated with this value. Occurs before any data references. Omitted if the value does not have a bookkeeping object.

EndOfBufr   24U

Fields: none

Meaning: end of current block, go to next

NOP   0xFFU

Fields: none

Meaning: NOP or filler; skipped.

Additional Table Of Contents Issues

The description above covers the byte level format of the TOC. However, there are a number of fine points that are necessary to understand how the TOC is used.

Standard Object IDs

Object ID 0 is never used, to avoid confusion with NULL, and for error checking.

The first 2^{16} object IDs (out of 2^{32}) are reserved for standard objects specified in the Bento specification. Thus, any ID with the two high order bytes 0 is the ID of a standard object. Most standard objects are type or property description objects, but some are specific objects required by the implementation. A list of the currently defined standard objects is given in Appendix D.

Standard objects are intended for use in defining the format of containers and supporting the API, and are not intended to be used for application data, types, or properties. Standard objects are required to "bootstrap" the API, since even finding the global name in a type description requires knowing a particular property and type.

TOC entries for standard objects are not required, but they are permitted. Thus, the IDs for standard objects may be used without a corresponding TOC entry.

Any given container can contain TOC entries for any subset of the defined properties for a given standard object, from none to all. If the TOC **does** contain entries for defined properties of standard objects, the entries must conform to the specification. Thus, a reader need not look in the TOC when it encounters a standard object ID.

Allowing entries for standard objects in the TOC gives us a mechanism for providing some backward compatibility. If we want to introduce a new standard object, we can put its description in the TOC of backward compatible containers for a while. Then older applications can still understand that object, assuming they can understand the description. At a minimum, the object description can be used to explain to the user what's missing and what she can do about it.

Additional "non-standard" properties can be given for standard objects, as long as they do not conflict with the standard properties.

TOC Self Reference

Every TOC contains a standard object that is used to describe the TOC itself. In particular, it is object ID 1, so the TOC entries for the TOC itself always come at the beginning of the TOC.

Additional TOC properties can be useful. For example, an index to speed access to the entries by ID could be attached to the TOC through another property. Potentially several such indexes, using different formats, could be attached.

Applications are free to attach additional properties to object 1. For example, it is sometimes useful to have an array of "root" objects in a container, from which all other objects can be found. Information about when the last update was performed, by whom, etc. could be maintained using properties of object 1. Etc.

Generation of Object IDs

Object IDs other than IDs of standard objects are generated by sequentially incrementing a counter from `0x00010000`. Object IDs are never reused in later generations of a container if an object is deleted. The last ID number generated is kept as a property of object # 1 to allow generating further IDs without reuse.

Generation Number

The current generation number of a Bento container is the generation number of its TOC, as recorded in the TOC value entry. Any change at all in a container requires a change to the TOC (since at least one entry has to change), so this is logically consistent.

Immediate values

Eight bytes to twelve bytes of an entry are required to "point to" the value data for an entry (four or eight bytes of offset and four bytes of length). When the property value is large, such as a video clip, PICT, or rich text object, this is reasonable, but when it is just an object ID or an integer, it is excessive.

An entry can be an immediate value. In this case it contains data instead of a offset and length referring to the data. All of the other fields of the entry are interpreted exactly as before. In particular, the type of the entry still describes the format of the value.

The library automatically creates immediate values when a four or less bytes of data are written to a new value segment. If more data is written to the value later, the API automatically converts it to a normal TOC entry.

Note that it is perfectly legal for an entry to "point to" a single byte of data, or even zero bytes of data, somewhere in the file. Such an entry will never be created by the library, but will be read correctly if it is encountered. Immediate values are only an optimization and are never required.

Spelling of Globally Unique Names

A given globally unique name is the value of a property of a type, property, or other description. The type of that property entry defines the format of the name. Currently, all Globally Unique Names must be in ISO 9070 name syntax, and must be derived from a name provided by an ISO naming authority.

Updating and Bookkeeping Information

The information used by the library to record updates and to perform reference tracking is stored as normal values in objects. Of course, these values are marked with standard properties known to the library.

The format of this information is not described in this version of the Bento specification. It is fully documented in the library source code.

Format Usage Issues

Dealing With Tagged Streams

The actual value data corresponding to a TOC entry in general will be completely "naked," with no required bytes specified by the format around it at all. This is in contrast to "internally tagged" file formats, in which each chunk of data is required to begin with (at least) a size and type, and possibly other information.

To accomodate existing internally tagged data formats, in Bento a given format of value can be defined to contain its own size, type or any other metadata. Furthermore, a TOC can be created that provides different TOC entries for data and tags in a stream of tagged chunks. In this case the tags are still in the stream, but they have been "logically separated" from the data when viewed through the Bento API. The tags can even be omitted from the TOC altogether, in which case they will be invisible if the data is viewed through the API.

Chapter 8: Format Usage

This chapter serves two purposes. FIrst, it provides several examples of how the container format can be used. These are complete, worked examples with specific values for all the relevant TOC entries.

Second, the more advanced examples show how to use the built-in extensibility of Bento to address additional needs, by defining the required properties and types. Thus these advanced examples provide examples of how to extend the basic format.

Usage Examples _____

These usage examples are given in a relatively complete and literal form. Even the TOC entries for standard objects are given, although they are not required, to provide a complete picture of what is going on.

The examples are given in the prevous, tablular form of the TOC, which is much more human readable than the new stream format.

Many details of the representation have been invented for these examples, such as the actual ID values, details of the naming conventions used in the property and type names, etc. The discussion in the Format Overview and Format Definition chapters explains the items which are part of the standard as such.

In particular, the Global Unique Names in the examples have not been revised to conform to ISO structured name syntax.

Embedded Stream Files

Suppose we have a container that simply contains two objects, with references from the first stream to the second. For example, the first object could consist of a stream of rich text, and the second object could be an image that is logically embedded in the text. Let us further suppose that the image has two alternate representations in different formats, each of which is a stream. This example exercises much of the basic machinery of Bento.

Below is a picture of the relevant parts of the TOC, and following that is a detailed discussion of the entries in the TOC.

. . .

a	268	18	22	vo.1	vl.1	0	0

. . .

b	400348	18	22	vo.2	vl.2	0	0
c	400563	18	22	vo.3	vl.3	0	0
d	723421	38	268	vo.4	vl.4	6	0
e	723655	38	400348	vo.5	vl.5	8	0
f	723655	38	400563	vo.6	vl.6	11	0

. . .

> vo.1: "Bento:ContentStandards:Text:RTXT"
> vo.2: "AppleComputer:Imaging:PICT"
> vo.3: "Microsoft:Imaging:WindowsMetafile"
> vo.4: <the actual RTXT stream...embedded ID 723655...>
> vo.5: <the actual PICT stream>
> vo.6: <the actual Windows metafile stream>

Each TOC entry is labeled with a lowercase letter to make it easier to refer to it; the letters are not actually part of the TOC contents. For the sake of this example we assume that the rich text type is a standard object that does not need to be in the TOC, but that an entry is provided anyway. We also assume that the types of the image streams are not standard objects, so that they must be provided.

Entry (a) is the only entry in the rich text type description. The local ID of the rich text type is 268. Note that since this is less than 2^{16} we know that it is a standard object. Thus this TOC entry is not really required, but it is provided for illustative purposes. The value of the entry is simply the Globally Unique Name that identifies the type. The text of the Globally Unique Name is listed below the TOC entries, labeled "vo.1" (for "value offset 1"). The value of vl.1 is the length of the string, including the null at the end (I didn't count). The property ID (18) means "TypeName"; the type ID (22) means "GloballyUniqueName". The description object is immutable, so it has generation 0. No flag bits are set (in this example, we are not using any flags).

Entries (b) and (c) are very similar. Each one is a type description object. Because their IDs are greater than 2^{16}, we know that they are not standard objects, and thus they are actually required in the TOC (this is mainly for illustrative reasons). The property and type IDs are the same as in entry (a).

Entry (d) is the single entry for the RTXT object. Its property ID (38) means "Primary Value". Its type ID (268) is the same as the ID of entry (a), indicating that its value is an RTXT value. Its offset (vo.4) points to the actual rich text data, which contains an embedded ID (723655). Its generation number (6) means that its entry and value were last modified in the sixth generation (copy) of this container.

Entries (e) and (f) are also very similar, but in a different way than (b) and (c). Note that both entries have the same ID, indicating that they are both properties of the same object. Furthermore, both entries have the same property ID (38) indicating that they are both primary values of the object. Thus, they are alternative representations of the primary value. Looking at their type IDs, we can see that one value is a PICT (400348) and the other is a Windows metafile (400563). Furthermore, looking at the generation numbers, we can see that both have been updated more recently than the text stream but that the Windows metafile was updated most recently.

The TOC Itself

A somewhat more recursive example is the description of the TOC by itself. Every TOC actually contains such a self-description, so reader code actually has to deal with some of this structure, but it will typically not be visible at the application level.

This is a portion of the same TOC as the previous example, so we will see some relationships between the content objects described above, and the properties of the TOC itself.

a	1	38	5	vo.1	vl.1	11	0
b	1	2	6	723689	–	11	1
c	1	3	25	723421	–	6	1
d	2	20	22	vo.2	vl.2	0	0
e	3	20	22	vo.3	vl.3	0	0
f	5	18	22	vo.4	vl.4	0	0
g	6	18	22	vo.5	vl.5	0	0

. . .

vo.1: <the actual TOC itself>

vo.2: "Bento:Basic:TOC:IDSeed"

vo.3: "Bento:Basic:TOC:RootContentObject"

vo.4: "Bento:Basic:TOC:AbsoluteTOCFormat"

vo.5: "Bento:Basic:TOC:Integer4Byte"

Entry (a) is an actual reference to the TOC as a value. The offset field (vo.1) will contain the offset of the TOC in the container(that is, the offset of the beginning of this entry). The length field (vl.1) will contain the length of the TOC in bytes. The property indicates that this is the primary value for the TOC object, and the type indicates that it is the normal top-level TOC format.

Entry (b) is also a property of the TOC object. It contains a value 1 larger than the last ID used. Note in this case that it is somewhat higher than highest ID that appeared in the previous example. This occurs if more objects were created, and then deleted. The seed prevents reuse of those IDs, in case some of them have been remembered (either within this container, or in external references to objects in this container). This prevents accidental aliasing.

Since the value of entry (b) is only four bytes long, it can be stored as an immediate value. Note that the immediate flag is set.

Entry (c) indicates the root object of the content in the container. Note that it contains the ID of the RTXT stream in the previous example, since that is the root content object. The ID is also an immediate value.

Note that the generation numbers of the TOC entries correspond to generation numbers of the relevant content objects. The generation of the TOC value itself is the generation of its most recent object. The generation of the root reference, however, indicates when the root was set to that object. It is the same generation as the object itself, so probably that object has been the root since it was created.

The remaining entries, (d) through (g), would not actually appear in a normal TOC because they are standard objects. Note, however, that they would be legal. They are provided for illustrative purposes. The first two are property descriptions, and the second two are type descriptions. Note that the only property of each description is the Globally Unique Name.

Types and Properties

Our last usage example is the most recursive, and would never occur in a real Bento container, but it documents the format, and it may be interesting as an extreme case. The example shows the top of thetype hierarchy, where the properties TypeName and PropertyName, and the type GloballyUniqueName are defined. Naturally these descriptions are all standard objects, and they are very unlikely to appear in any TOC. Furthermore, realistically, if they did, no reader would be able to use them. However, as an example, this may give some insight into both the more exotic uses of Bento, and the structure of the type and property mechanism.

In addition, to tie the example to some of the concrete types we have already seen, we give a more complete derivation of the RTXT type, including its supertype COBJ, and a dictionary that describes RTXT in terms of COBJ.

. . .

a	18	20	22	vo.1	vl.1	0	0
b	20	20	22	vo.2	vl.2	0	0
c	21	20	22	vo.3	vl.3	0	0
d	22	18	22	vo.4	vl.4	0	0
e	22	21	25	26	–	0	3
f	25	18	22	vo.5	vl.5	0	0
g	26	18	22	vo.6	vl.6	0	0
h	76	20	22	vo.7	vl.7	0	0
i	132	18	22	vo.8	vl.8	0	0
j	136	18	22	vo.9	vl.9	0	0
k	268	18	22	vo.10	vl.10	0	0
l	268	76	136	vo.11	vl.11	0	0
m	268	21	25	132	--	0	3

. . .

vo.1: "Bento:Basic:TypeName"
vo.2: "Bento:Basic:PropertyName"
vo.3: "Bento:Basic:SuperType"
vo.4: "Bento:Basic:GloballyUniqueName"
vo.5: "Bento:Basic:LocalIDReference"
vo.6: "Bento:Formats:Printable7BitAscii"
vo.7: "Bento:Descriptions:DataFormat"
vo.8: "Bento:Formats:COBJ:GenericStream"
vo.9: "Bento:Formats:COBJ:TypeDictionary"
vo.10: "Bento:ContentStandards:Text:RTXT"
vo.11: <the actual RTXT type dictionary>

In entries (a), (b), and (d) we finally see the definition of the ubiquitous property IDs 18, 20, and 22. Aside from the fact that they use each other, and (b) and (d) use themselves, they are fairly normal property and type descriptions.

Entry (c) introduces the SuperType property, and it is used in entry (e). Entry (e) effectively says that a GloballyUniqueName is spelled in printable 7 bit ASCII. Note that it uses an immediate reference to the supertype. Entry (f) defines the local ID type we have been using, and entry (g) defines printable 7 bit ASCII.

Entry (h) introduces a very general property that allows us to attach data format descriptions to objects. It is intended primarily for use in type descriptions. The interpretation of the data format description will depend on its type.

Entries (i) and (j) are parts of the description of COBJ, a stream oriented data definition standard under development at Apple. (i) describes the type of actual COBJ values, while (j) describes the type of COBJ type dictionaries, which define the format of particular streams.

Finally, entries (k) and (l) and (m) describe RTXT (ID 268, as we saw in the first example). Here in addition to the Globally Unique Name of RTXT, we have the COBJ type dictionary, and the reference to COBJ as the supertype of RTXT.

Multi-Media Issues _____

Multi-media formats require two types of support beyond what is defined in the basic format::

1)   They need to be able to interleave values and then reconstitute them in two ways:

a)   by playing them linearly, without holding a large table of contents in memory or seeking off to look at the table of contents, and

b)   by asking for them as "normal" values, and getting them either as a single hunk of data in a buffer or as a non-interleaved stream.

2)   They need to have a stream of data containing **local** tables of contents, which will (typically) describe what is coming up in the stream, allowing a "player" to get ready, select the right stuff as it comes along, etc.

Using interleaved values

Requirement (1) is largely handled by value segments, which permit arbitrary interleaving of values. This only addresses the format flexibility and the ability to retrieve the complete value as a "normal" value.

Note that the actual interleaving is determined by the application that writes out the values into the container. Thus, the container does not "understand" interleaving, and has no built in mechanism to define the particular interleaving chosen. This allows total control by the application.

To play interleaved values efficiently, without frequent references to a remote table of contents, the player application and the authoring tools need an additional contract. Each interleaved object needs to "know" the rules by which its values are broken up, how far apart the pieces are, etc. This information can be recorded in an additional property attached to each interleaved object.

Using this additional information, the player application can read the values directly from a stream, without having to remember the TOC entry for each partial value. The player will only need to reference a table of contents to select objects to play, and to retrieve the interleaving specifications.

The definition of particular interleaving is beyond the scope of this specification If and when an industry standard interleaving specification emerges, we can certainly make sure Bento meshes with this standard.

Thus, all that is required to address requirement (1) is an interleaving specification attached to the appropriate objects.

<u>Local tables of contents</u>

Problem (2) can be solved by allowing partial tables of contents in a container. This is required for other reasons, such as breaking a Bento container across multiple volumes. This question is currently under investigation.

Other Usage Issues_____

<u>Property Index</u>

Sometimes we want to find all the objects in a container with a particular (set of) value(s) of a particular property. With a few objects that have the property in a sea of irrelevant ones that do not have the property, such a search may become very expensive.

To facilitate finding all the objects with a given property, we can keep an index of objects according to which properties they support.

<u>External Reference Table</u>

When a container is moved into a new environment, often its links to other containers, files, etc. need to be fixed up. This fixup can be done by a general utility, or even by the finder (in some future version) if the links are sufficiently inspectable.

The format of the TOC as described provides some of this inspectability. All references to external entities are indirect. Therefore, external references can be found by a scan of the TOC. However this remains somewhat inconvenient. It forces utilities to distinguish between external references and other uses of indirect values by inspecting their types. Furthermore, it does not specify any particular scheme for sharing the potentially common information in external references, such as directory path names.

To simplify the task of fixing up the links of a container, we can provide a list of all the external references in the TOC, as a property of the TOC itself. In its simplest form the external reference list only needs to contain the object IDs of all objects with external references. However, we will probably want some information about the shared content of these indirect values as well, to avoid redundant binding.

<u>Encoding</u>

We need a way to specify encoding information for the value of each entry. A single object may have different property values created on different systems (for example, a PICT and a Windows metafile), and the encoding information needs to be able to reflect this.

The encoding information is logically independent of the type of the value, but in many cases, the encoding may be derivable from the type. In some cases, however, the type and encoding are essentially independent.

To deal with this set of design constraints, we will specify encoding by the type. However, when we have a type that is essentially independent of encoding, we will need to specify the encoding explicitly as a property of the type description. Note that in some cases we could have two or more types with different encoding properties, but the same underlying "abstract" format type.

Appendix A: API Summary

Types and Constants _____

Low level basic types

```
typedef char      CHAR;

typedef unsigned char     UCHAR;

typedef short     SHORT;

typedef unsigned short    USHORT;

typedef long      LONG;

typedef unsigned long     ULONG;
```

Types

```
typedef struct                    CMSession_ *CMSession;

typedef struct                    CMContainer_ *CMContainer;

typedef struct                    CMObject_ *CMObject;

typedef CMObject                  CMProperty;

typedef CMObject                  CMType;

typedef struct                    CMValue_ *CMValue;

typedef CHAR                      *CMOpenMode;

typedef CHAR                      *CMGlobalName;

typedef CHAR                      *CMErrorString;

typedef CM_CHAR                    *CMMetaData;

typedef void                      *CMRefCon;

typedef void                       *CMPtr;

typedef UCHAR                     *CMMagicBytes;

typedef CM_UCHAR                  *CMDataPacket;

typedef CM_UCHAR                  *CMDataBuffer;

typedef CM_UCHAR                  *CMPrivateData;

typedef CM_UCHAR                  CMReference[4];

typedef UCHAR                     CMSeekMode;

typedef UCHAR                     CMBoolean;

typedef USHORT                    CMContainerUseMode;

typedef USHORT                    CMContainerFlags;
```

```
typedef CM_USHORT                      CMContainerModeFlags;

typedef USHORT                         CMEofStatus;

typedef CM_LONG                        CMErrorNbr;

typedef ULONG                          CMGeneration;

typedef ULONG                          CMSize;

typedef ULONG                          CMCount;

typedef void                           *CMPtr;

typedef void                           ( *CMHandlerAddr)();

typedef  CMHandlerAddr ( *CMMetaHandler)
                   (CMType,
                   const CMGlobalName);
```

Constants

```
const CMContainerUseMode          kCMReading = 0x0001

const  CMContainerUseMode         kCMWriting = 0x0002

const CMContainerUseMode          kCMReuseFreeSpace = 0x0004

const CMContainerUseMode          kCMUpdateByAppend = 0x0008

const CMContainerUseMode          kCMUpdateTarget = 0x0010

const CMContainerUseMode          kCMConverting = 0x0020;

const CMSeekMode                  kCMSeekSet = 0x00;

const CMSeekMode                  kCMSeekCurrent= 0x01;

const CMSeekMode                  kCMSeekEnd = 0x02;
```

Operation Definitions _____

Session Operations

```
CMSession  CMStartSession(CMMetaHandler metaHandler,
                   CMRefCon sessionRefCon)

void  CMEndSession(CMSession sessionData,
                   CMBoolean closeOpenContainers)

void  CMAbortSession(CMSession sessionData);

CMRefCon   CMGetSessionRefCon(CMContainer container)

void  CMSetSessionRefCon(CMContainer container,
                   CMRefCon refCon)
```

```
CMHandlerAddr  CMSetMetaHandler(const CMSession sessionData,
                    const CMGlobalName typeName,
                    CMMetaHandler metaHandler)

CMHandlerAddr  CMGetMetaHandler(const CMSession sessionData,
                    const CMGlobalName typeName)

CMHandlerAddr CMGetOperation(CMType targetType,
                    const CMGlobalName operationType);
```

Container Operations

```
CMContainer  CMOpenContainer(CMSession sessionData,
                    CMRefCon attributes,
                    const CMGlobalName typeName,
                    CMContainerUseMode useFlags);

CMContainer  CMOpenNewContainer(CMSession sessionData,
                    CMRefCon attributes,
                    const CMGlobalName typeName,
                    CMContainerUseMode useFlags,
                    CMGeneration generation,
                    CMContainerFlags containerFlags,
                    ...);

void CMGetContainerInfo(const CMContainer container,
                    CMGeneration *generation,
                    CMContainerFlags *containerFlags,
                    CMGlobalName typeName);

CMSession  CMGetSession(CMContainer container)

VOID CMCloseContainer(CMContainer container);

VOID CMAbortContainer(CMconst_CMContainer container);
```

Type and Property Operations

```
CMType CMRegisterType(CMContainer targetContainer,
                    const CMGlobalName name);

CMProperty CMRegisterProperty(CMContainer targetContainer,
                    const CMGlobalName name);

CMBoolean CMIsType(CMObject theObject);

CMBoolean CMIsProperty(CMObject theObject);

CMType CMGetNextType(CMContainer targetContainer,
                    CM CMType currType);

CMProperty CMGetNextProperty(CMContainer targetContainer,
                    CMProperty currProperty);
```

```
CMCount  CMAddBaseType(CMType type,
                    CMType baseType)

CMCount  CMRemoveBaseType(CMType type,
                    CMType baseType)
```

Object Operations

```
CMObject CMNewObject(CMContainer targetContainer);

CMObject CMGetNextObject(CMContainer targetContainer,
                    CMObject currObject);

CMProperty CMGetNextObjectProperty(CMObject theObject,
                    CMProperty currProperty);

CMObject CMGetNextObjectWithProperty(CMContainer targetContainer,
                    CMObject currObject,
                    CMProperty property)

CMContainer CMGetObjectContainer(CMObject theObject);

CMGlobalName CMGetGlobalName(CMObject theObject);

CMRefCon  CMGetObjectRefCon(CMObject theObject)

void  CMSetObjectRefCon(CMObject theObject,
                    CMRefCon refCon)

VOID CMDeleteObject(CMObject theObject);

VOID CMDeleteObjectProperty(CMObject theObject,
                    CMProperty theProperty);

VOID CMReleaseObject(CMObject theObject);
```

Value Operations

```
CMCount CMCountValues(CMObject object,
                    CMProperty property,
                    CMType type);

CMValue CMUseValue(CMObject object,
                    CMProperty property,
                    CMType type);

CMValue  CMGetNextValue(CMObject object,
                    CMProperty property,
                    CMValue currValue)

CMValue CMNewValue(CMObject object,
                    CMProperty property,
                    CMType type,
                    ...);
```

```
CMValue  CMVNewValue(CMObject object,
                     CMProperty property,
                     CMType type,
                     va_list dataInitParams)

CMSize CMGetValueSize(CMValue value);

CMSize CMReadValueData(CMValue value,
                     CMPtr buffer,
                     CMCount offset,
                     CMSize maxSize)

void CMWriteValueData(CMValue value,
                     CMPtr buffer,
                     CMCount offset,
                     CMSize size)

VOID CMInsertValueData(CMValue value,
                     CMPtr buffer,
                     CMCount offset,
                     CMSize size)

VOID CMDeleteValueData(CMValue value,
                     CMCount offset,
                     CMSize size)

VOID CMDefineValueData(CMValue value,
                     CMSize offset,
                     CMSize size);

void  CMMoveValue(CMValue value,
                     CMObject object,
                     CMProperty property)

VOID CMGetValueInfo(CMValue value,
                     CMContainer *container,
                     CMObject *object,
                     CMProperty *property,
                     CMGeneration *generation);

void  CMSetValueType(CMValue value,
                     CMType type)

void  CMSetValueGeneration(CMValue value,
                     CMGeneration generation)

void CMDeleteValue(CMValue value);

void CMReleaseValue(CMValue value);
```

Reference Operations

```
CMReference    *CMNewReference(CMValue value,
                    CMObject referencedObject,
                    CMReference theReferenceData)

CMObject  CMGetReferencedObject(CMValue value,
                    CMReference theReferenceData)

CMReference   *CMSetReference(CMValue value,
                    CMObject referencedObject,
                    CMReference theReferenceData);

void CMDeleteReference(CMValue value,
                    CMReference theReferenceData);

CMCount CMCountReferences(CMValue value);

CMReference *CMGetNextReference(CMValue value,
                    CMReference currReferenceData);
```

Appendix B: Handler Interfaces

This appendix documents the handler mechanism, and the prototypes of a number of important handler operations.

The Bento sources come with a complete and well-documented set of example handlers that run in most environments, since they depend only on the ANSI C libraries. Many of the points in this appendix will be easier to understand if you read it in conjunction with the code.

Meta-Handler Interface

The handlers registered with Bento are actually metahandlers, because they are not called directly to carry out the operations. Instead, they are called to get procedure pointers to specific handlers that can carry out the desired operation. Typically, these procedure pointers will be cached and then used in the normal manner.

Each metahandler may provide handlers for any number of operations.

Each metahandler implements only one operation, with the following prototype:

```
CMProcPtr CMMetaHandler(CMType targetType,
                  const CMGlobalName operationType);
```

This is the required prototype of any metahandler registered by the application using `CMSetHandler`. When a specific operation is required, the `meta`handler is called, and it must return a `CMProcPtr` for the operation, or return `NULL` to indicate that the operation is not available. Once retrieved, the `CMProcPtr`s may be cached indefinitely.

`targetType` is the refnum of the type to which the operation will be applied, and `operationType` is the name of the desired operation. `targetType` is required because in some cases the operation may be applied to values whose type has no global unique name.

This approach provides more flexibility than simply passing a vector of procPtrs, and allows each operation to have its own prototype for static type checking, which would be impossible if operations were indicated by passing a selector.

Thre are three varieties of metahandlers: session, container, and value.[1] Each of these varieties is expected to provide certain operations, as documented below.

Session Handler Operations

There are three handlers that must be provided by the metahandler passed in to `CMStartSession()`. They should have the following prototypes

---

[1]   Note that this is not a fundamental distinction. In principle, a single metahandler could function in two or even all three of these roles. However, as a practical matter, this would not be a good way to structure things.

```
void error_Handler(CMErrorNbr errorNumber, ...)
```

The error reporting handler is a required special routine whose address is asked for by `CMStartSession()`. All errors are reported through here. Using the API routine `CMError()`, calls can be made outside the Container Manager as long as the session is defined (`CMStartSession()`). The intent is that handlers will call `CMError()` to report their errors just like the Container Manager does internally.

The API generally assumes the error handler will **never** return. It tries to protect itself in case you do, but don't count on it! Assume your container is screwed if this handler is called.

The Container Manager API makes available some of the same routines used internally. Specifically, the ability to take an string that can contain inserts and "edit in" those inserts (`CM[V]AddMsgInserts()`). See the utility routines documented at the end of this appendix.

```
void * Alloc_Handler(CMSize size);
```

The Container Manager API requires some form of memory management that can allocate memory and return pointers to it. By generalizing this as a handler you are free to choose a memory management mechanism appropriate to your environment.

If you are running in a standard C runtime environment, mapping this handler directly onto the C runtime `malloc()` may prove sufficient.

```
void Free_Handler(CMPtr ptr);
```

The Container Manager API calls this handler when it wants to free up memory it no longer needs. The memory attempting to be freed will, of course, be memory previously allocated by the memory allocator handler.

Container Handler Operations

Bento requires container metahandlers to provide certain operations. Bento implements all of the other operations in the API using these required operations.

Operations involving updating may be difficult to understand without reading the code.

Container handlers must provide the stream operations. These have interfaces and semantics derived from the standard C library stream operations:

```
VOID CMOpenStream(CMValue value,
                  CMStreamMode mode);

VOID CMCloseStream(CMValue value);

CMSize CMReadStream(CMValue value,
                    CMPtr buffer,
                    CMSize elementSize,
                    CMCount theCount);
```

```
CMSize CMWriteStream(CMValue value,
                     CMPtr buffer,
                     CMSize elementSize,
                     CMCount theCount);

CMSize CMGetPosStream(CMValue value);

CMSize CMSetPosStream(CMValue value,
                      CMSize posOff,
                      CMSeekMode mode);

VOID CMFlushStream(CMValue value);

CMStreamStatus CMEofStream(CMValue value);
```

When the Bento library calls the stream operations to perform I/O on a container, the value it passes as the first argument to the operations is the referenceConstant originally provided when the container was Used.

```
VOID CMReadLabel(void *attributes,
                 CMMagicBytes magicBytesSequence,
                 CMContainerFlags *containerFlags,
                 CMEncoding *encoding,
                 USHORT *majorVersion,
                 USHORT *minorVersion,
                 CMSize *tocOffset,
                 CMSize *tocSize);
```

This operation finds the label of the container specified by attributes and returns all of the information it contains. The location of the label is container type dependent. However, all files must have a label as specified in the format definition.

```
VOID CMWriteLabel(void *attributes,
                  CMValue value,
                  CMMagicBytes magicBytesSequence,
                  CMContainerFlags containerFlags,
                  CMEncoding encoding,
                  USHORT majorVersion,
                  USHORT minorVersion,
                  CMSize TOCOffset,
                  CMSize TOCSize);
```

This operation is called after all of the necessary information has been written to the container by Bento, just before it closes the container. It must write the label for the container (if any) and do any implementation dependent container writes. When this operation completes, the state of the container should be such that if the next call on the handler is a CMCloseStream, the container will be well formed and usable if it is opened again in the future.

```
CMValue CMReturnParentValue(CMRefCon refCon);
```

This handler routine is used **only** for embedded containers. It is called at open time by `CMOpen[New]Container()` so that the Container Manager may determine for itself that it is opening an embedded container for a value and what that value is. It is the parent `CMValue` for this handler that is retured by this function.

This is a **required** handler routine for embedded containers only. Therefore, container metahandlers will typically return `NULL` when asked for this handler.

```
static CM_UCHAR *returnContainerName_Handler(CMRefCon refCon)
```

When the Container Manager reports errors it passes appropriate string inserts that may be formatte into the error message. One of those inserts is usually an string that identifies for which container the error applies. The handler defined here is used by the Container Manager to get that identification.

For files, the most appropriate identification is typically the pathname for the container file.

This is an **optional** routine for reading and writing. If not provided, the type name passed to `CMOpen[New]Container()`, i.e., the metahandler type, is used for the identification.

```
CMType CMReturnTargetType(CMRefCon refCon,
                          CMContainer container);
```

If `CMOpenNewContainer()` is called with useFlags set to `kCMUpdateTarget`, then the intent is to open a new container which will record updating operations of updates applied to **another** distinct (target) container. The target container is accessed indirectly through a dynamic value whose type is gotten from a this handler. This handler must have be supplied and it must return a type which will spawn a dynamic value when `kCMUpdateTarget` is passed to `CMOpenNewContainer()`.

The process of creating a dynamic value (by the Container Manager using the returned type) will generate a "real" value in the parent container (the new container to record the updates). That value can be used by future `CMOpenContainer()`'s to "get at" the target again. To be able to find it, the created value becomes a special property of TOC #1. `CMOpenContainer()` will look for that property.

This handler is running in the context of the parent container, which is passed as a parameter along with the usual refCon. The handler registers the type in this container.

See the description of dynamic values and their base types for further details on how these types spawn dynamic values.

This is a **optional** routine for reading and writing. It is **required** for updating when `CMOpenNewContainer()` is called with useFlags set to `kCMUpdateTarget`.

```
void FormatData(CMRefCon refCon,
                CMDataBuffer buffer,
                CMSize size,
                CMPrivateData data);
```

This handler is used to format "internal" Container Manager data to be written to the container (e.g., updating data). 1, 2, or 4 bytes (size 8-bit byte) "chunks" of data are expected to be copied from the data to a buffer. Pointers to the data and the buffer are passed in. The buffer can always be assumed large enough to hold the data. The pointer to the data can be assumed to point to a `CM_UCHAR` if size is 1, `CM_USHORT` if size is 2, and `CM_ULONG` is size if 4.

The 1, 2, or 4 bytes are, of course, stored in the `CM_UCHAR`, `CM_USHORT`, or `CM_ULONG` as a function of the architecture. These may be a different size than what is expected to be written to the container. Indeed, it is the potential difference between the architecture from the data layout in the container that this handler must be provided.

The information is stored in the container in a layout **private** to the Container Manager. For example, it is used to format the fields of the TOC. The library does repeated calls to this handler to format the information it needs into a buffer that is eventually written using the write handler.

In this example `CM_UCHAR`, `CM_USHORT` and `CM_ULONG` directly map into the container format 1, 2, and 4 byte entities. Hence the formatting is straight-forward.

This is an **optional** routine for reading and writing. It is **required** for writing or updating when `CMOpenNewContainer()` is called with useFlags set to `kCMUpdateTarget` or `kCMUpdateByAppend`.

```
void CMExtractData(CMRefCon refCon,
                   CMDataBuffer buffer,
                   CMSize size,
                   CMPrivateData data);
```

This handler is used to extract "internal" Container Manager data previously written to the container (e.g., updating data). 1, 2, or 4 bytes (size 8-bit byte) "chunks" of data are expected to be copied from a buffer to the data. Pointers to the data and the buffer are passed in. The buffer can always be assumed large enough to supply all the requested data. The pointer to the data can be assumed to point to a `CM_UCHAR` if size is 1, `CM_USHORT` if size is 2, and `CM_ULONG` is size if 4.

The 1, 2, or 4 bytes are, of course, formatted within the `CM_UCHAR`, `CM_USHORT`, or `CM_ULONG` as a function of the architecture. These may be a different size than what is expected to be written to the container. Indeed, it is the potential difference between the architecture from the data layout in the container that this handler must be provided.

This routine is used to store information in the container in a layout **private** to the Container Manager. For example, it is used to extract the fields of the TOC. The Container Manager does repeated calls to this handler to extract the information it needs from a buffer that it loads using the read handler.

This is a **optional** routine for writing. It is **required** for reading, or if an updating container is opened (i.e., a container used previously for updating). It is **required** for updating when `CMOpenNewContainer()` is called with useFlags set to `kCMUpdateTarget` or `kCMUpdateByAppend`.

Value Handlers

These are discussed in the chapter on Types and Dynamic Values.

Handler Support Routines _____

When writing handlers, certain facilities used by the API are needed to provide a consistent interface to the application.

Session Handler Pass-Throughs

The handlers need to use the session facilities for memory management and error reporting, since they are effectively running as part of the library. These routines provide access to those facilities.

```
void  CMMalloc(CMSize size, CMSession sessionData)
```

This routine provides a access path for the handler writer to use the same memory management allocator defined for the current Container Manager session. size bytes are allocated as defined by that handler.

The session memory allocator handler is defined by the metahandler passed to `CMStartSession()`. The `sessionData` is the current session refNum returned from `CMStartSession()`.

```
void  CMFree(CMPtr ptr,
                  CMSession sessionData)
```

This routine provides a access path for the handler writer to use the same memory management deallocator defined for the current Container Manager session. A pointer (ptr) which must have been allocated by `CMMalloc()` is passed to release the memory in the manner defined by the handler.

The session memory deallocator handler is defined by the metahandler passed to `CMStartSession()`. The `sessionData` is the current session refNum returned from `CMStartSession()`.

```
void CMError(CMSession sessionData, CMErrorString message, ...);
```

This routine provides an access path for a handler writer to use the same error reporter defined for the current Container Manager session. The session error reporting handler is defined by the metahandler passed to `CMStartSession()`. The `sessionData` is the current session refNum returned from `CMStartSession()`.

Note, as currently defined, the error reporting handler should not return to its caller. But if it does, `CMError()` will also.

Error Reporting Support

```
char *CMAddMsgInserts(char *msgString,
                      CMSize maxLength,
                      ...);
```

This routine takes the (error) message string in `msgString` and replaces all substrings of the form "^n" with the inserts from the variable arg list of insert strings. The function returns the edited `msgString` as its result.

The string buffer must be passed in `msgString`. Its max length is also specified but this must be a value less than or equal 1024 (not including delimiting null). The message will be truncated to fit in the buffer. The string pointer is returned as the function result.

The substring "^0" is replaced with the first insert. "^1" is replaced with the second insert, and so on. It is assumed that there are enough insert strings to cover all the "^n"s called for (not unlike `printf()`).

Note, the "^n"s in the message string do not have to be in ascending order. Thus "^1" could occur before "^0" in the `msgString`.

```
char *CMVAddMsgInserts(char *msgString,
                       CMSize maxLength,
                       va_list inserts);
```

This routine is the same as `CMAddMsgInserts()` above, except that the extra (inserts) arguments are given as a variable argument list as defined by the "stdarg" facility.

```
CMErrorString CMGetErrorString(CMErrorString errorString,
                       CMSize maxLength,
                       CMErrorNbr errorNumber,
                       ...)
```

This routine takes a defined Container Manager (error) message number and its corresponding insert strings and returns a (english) string corresponding to the message number with the inserts filled into their proper positions. It is assumed the error number and inserts were the ones reported to the error handler.

The string buffer must be passed in errorString. Its max length is also specified but this must be a value less than or equal 1024 (not including delimiting null). The message will be truncated to fit in the buffer. The string pointer is returned as the function result.

This routine is provided as a convenience to easily convert the error codes and their corresponding inserts to a printable string.

```
CMErrorString CMVGetErrorString(CMErrorString errorString,
                       CMSize maxLength,
                       CMErrorNbr errorNumber,
                       va_list inserts)
```

This routine is the same as `CMGetErrorString()` above, except that the extra (inserts) arguments are given as a variable argument list as defined by the "stdarg" facility.

```
char *CMReturnContainerName(CMContainer container);
```

Generally the errors reported are provided with at least one insert that identifies which container we're talking about. The wording of the messages defined for the Container Manager assume this identification insert. The identification takes the form of the container "name" which is obtained from a handler routine provided for that purpose.

This routine is provided to test if the container metahandler provides a name handler, and call the handler if it exists. If it doesn't exist the container type name is returned.

Note, this routine is available to handler writers so that they can generalize their error reporting/message routines and word their messages with the container identification.

Value Handler Support

```
CMCount   CMScanDataPacket(CMType type,
                     CMMetaData metaData,
                     CMDataPacket dataPacket, ...)
```

This routine is used by a dynamic value's "new value" handler to extract the fields of a data packet passed to it by the Container Manager. The data packet represents all the `CMNewValue()` "..." parameters for the type also passed to the "new value hander".

Only that portion of the `CMNewValue()` "..." parameters associated with the type are passed in the data packet. The Container Manager determines the parameters by the placement of the type within its heirarchy (types may have base types) and the metadata.

The Container Manager accesses the metadata through a "metadata" handler for the type to build the data packet. `CMScanDataPacket()` inverts the the operation and allows the "new value" handler to extract the data back into distinct variables. The "new value" handler can use its own "metadata" handler to pass to the `CMScanDataPacket()` routine to extract the data. Each `CMScanDataPacket()` "..." parameter must be a pointer; extracted data read from the data packet are stored into the locations pointed to by the pointers.

The function returns the number of data items extracted and assigned to the parameters. This could be 0 if metadata is passed as `NULL`, or if an error is reported and the error reporter returns.

```
CMValue   CMGetBaseValue(CMValue value)
```

Returns the base value for a dynamic value and `NULL` if the value is not a dynamic value. It is expected that this routine will only be called from dynamic value handlers. Indeed, this is enforced!

Appendix C: Error Messages

In the list of error code definitions below, the comment is the error message that is provided for the error code. The "^n"s (^0, ^1, etc.) in the comments show where the inserts would go. ^0 is the first insert, ^1 the second, and so on.

GenericMessage

   ^0

BadTocSize

   TOC index table size out of range (^0 <= range <= ^1)

NoSession

   Unable to allocate session (task) global data

NoHandler

   Cannot allocate space for handler for type \"^0\"

BadWriteUse

   Cannot do a CMOpenNewContainer(...^0...) with useFlags set for update-in-place

NoContainer

   Cannot allocate CCB space for type \"^0\"

UndefMetaHandler

   Metahandler was not defined for type \"^0\"

HandlerError

   Allocation problem while looking up handler for type \"^0\"

NullMetaPtr

   Null metahandler pointer for type \"^0\"

UndefRoutine

   Handler routine(s) missing for operation type(s): ^0

NoTOC

   Cannot allocate TOC for container \"^0\"

BadReadUse

   Cannot do a CMOpenContainer(...\"^0\"...) with useFlags set for ^1

BadMagicBytes

   Invalid magic byte sequence in label for container \"^0\" - expected \"^1\", got \"^2\"

BadVersion

   Invalid version fields in label for container \"^0\" - expected ^2.^3, got ^4.^5

BadTOCRead

   Incorrect byte length read while reading TOC for container \"^0\"

NoObjEntry

    Cannot allocate space for TOC object entry for container \"^0\"

MultDef

    Multiple definition for TOC object ^0 for container \"^1\"

NoPropEntry

    Cannot allocate space for TOC property entry for container \"^0\"

BadContinue

    Invalid continued value (inconsistent types) in container \"^0\"

NoValueEntry

    Cannot allocate space for TOC value entry for container \"^0\"

BadOffset

    Invalid offset or value (^0, ^1) for TOC value entry for container \"^2\"

UndefObjects

    There is (are) ^0 undefined object(s) in container \"^1\"

NoStrValue

    Cannot allocate space for constant value in container \"^0\"

DupBaseType

    Cannot add dup base type \"^0\" to type \"^1\" in container \"^2\"

BadTOCWrite

    Incorrect byte length written while writing TOC for \"^0\"

NotGlobalName

    Have global name tied to value of wrong type in containter \"^0\"

BadGNameWrite

    Incorrect byte length written while writing global name \"^0\" in container \"^1\"

DupGlobalName

    Duplicate global name definition for \"^0\" in container \"^1\"

MultGlblNames

    Object ^0 alread defined -- trying to redefine it for \"^1\" (container \"^2\")

NoGlobalName

    Cannot allocate space for global name \"^0\" in container \"^1\"

NoGNameLoad

    Cannot allocate space for global name during load in container \"^0\"

BadGNameRead

    Incorrect byte length read while reading global name in container \"^0\"

NotGName

Invalid global name string read in container \"^0\"

BadType

Invalid ^0 type passed to ^1 for container \"^2\"

2Containers

Objects not in same containers (containers \"^0\" and \"^1\")

3Containers

Objects not in same containers (containers \"^0\", \"^1\", and \"^2\")

MissingMinIDSeed

Min ID seed value missing in TOC object 1 in container \"^0\"

MissingTOCObj

TOC object ID 1 missing in TOC in container \"^0\"

NotConverting

Cannot use CMDefineValueData except for converting container \"^0\"

BadDefineData

Attempt to define offset (^0) out of range in container \"^1\"

BadValue

Attempt to use a deleted value in container \"^0\"

BadObject

Attempt to use a deleted object in container \"^0\"

BadContainer

Container for ^0 (\"^1\") does not belong to the container being used (\"^2\")

NoValue

No value defined for object ID ^0, property \"^1\" in container \"^2\"

HasValue

Cannot set a (sub)value to an already defined value (container \"^0\")

AlreadyReleased

Attempting to release a value already completely released in container \"^0\"

NotReleased

A dynamic value has not been fully released in container \"^0\"

MissingFreeTotal

Total free space value missing in TOC object 1 in container \"^0\"

DupType

Attempt to insert two values with the same type (^0) in container \"^1\"

HasNoValue

No value defined for CMReadValueData in container \"^0\"

BadWrite

    Write error writing to container \"^0\"

CantWriteGlbl

    Cannot write to a global name in container \"^0\"

Offset2Big

    Write/insert offset (^0) beyond end of a value in container \"^1\"

Internal1

    Internal error! Unknown flags setting (0x^0)

MissingIDSeed

    ID seed value missing in TOC object 1 in container \"^0\"

AmbiguousType

    ^0 object is ambiguous in container \"^1\"

TypeNotGlobal

    ^0 object is not for a global name in container \"^1\"

MissingFreeList

    Internal error - empty free list for property in container \"^0\"

NoNewValueHandler

    A \"new value\" handler is not defined for type \"^0\"

UndefReference

    Object ID ^0 from a reference is undefined in container \"^1\"

BadObjectKind

    Invalid kind of object -- expected ^0 in container \"^1\"

WriteIllegal1

    Cannot write to a container (\"^0\") opened for reading

WriteIllegal2

    Attempt to write to a protected object in container \"^0\"

ReadIllegal

    Cannot read from a container (\"^0\") opened for writing

MissingSize

    Size value missing in TOC object 1 in container \"^0\"

BadSize

    Inconsistent size values between container label and TOC (container \"^0\")

Internal2

    Internal error! TOC offset to offset/size value not found in container \"^0\"

CantDelete1

Attempt to delete to a protected object in container \"^0\"

CantDelete2

Attempt to delete to a property with a protected value in container \"^0\"

CantDelete3

Attempt to delete to a protected value in container \"^0\"

StillOpen

Container \"^0\" is still open at session close time

EmptyRead

Cannot read from an empty embedded container value (container \"^0\")

NoEmbedding

Cannot allocate space to save embedding status while opening container \"^0\"

BadGenNbr

Invalid generation number (^0) passed to ^1 (container \"^2\")

NoRef

Connot allocate space for an object reference in container \"^0\")

CantGetBase

CMGetBaseValue() may only be called from a dynamic value handler in container \"^0\"

MultTypeProp

Attempt to register a ^0 name (^1) in container \"^2\" -- already defined as a ^3

NotSameMode

Embedded container (\"^0\") must be opened in same mode as its parent (\"^1\")

CantDelete4

Cannot delete a value currently in use in container \"^0\"

MissingHandler

Memory allocator and dealocator handlers must be supplied

NoMissingBuffer

Unable to allocate private temporary buffer while opening type \"^0\"

MoveIllegal

Cannot move a value in a container (\"^0\") not opened for writing

DeleteIllegal

Attempt to delete ^0 in a container (\"^1\") not opened for writing

BadDefineType

Attempt to define additional data for a immediate value in container \"^0\"

NoExtensions

Cannot allocate space for TOC dynamic value entry for type \"^0\" in container \"^1\"

HandlerRecursed

Attempt to use dynamic value handler for ^0() recursively in container \"^1\"

BadRealValue

Invalid base (\"real\") value passed to ^0() in container \"^1\"

NoMetahandler

A value operations metahandler is not defined for type \"^0\" in container \"^1\"

NotDynValue

A \"use value\" metahandler did not create a dynamic value in container \"^0\"

NoGlobalTable

Cannot allocate globally unique name table for container \"^0\"

BadMetaSpec

Invalid metadata format specification (%^0) for type \"^1\" in container \"^2\"

NoDeletesList

Internal error - empty \"deletes\" list for property in container \"^0\"

NoDataPacket

Cannot allocate space for data packet for type \"^0\" in container \"^1\"

BaseRelAttempted

A dynamic value release handler attempted to release its base in container \"^0\"

NoDynMetahandler

\"Use value\" handler for type \"^0\" in container \"^1\" MUST return a metahandler

MissingTotalSize

Total container size value missing in TOC object 1 in container \"^0\"

Internal3

Internal error! TOC offset to container value not found in container \"^0\"

AmbiguousUseFlags

Ambiguous updating useFlags passed to CMOpenNewContainer (...\"^0\"...) - ^1

NoTypeHandler

Dynamic value type handler not defined for updating in container \"^0\"

NotDynamicValue

Dynamic value in container \"^0\" not created to access target for updating

NoMetaDataHandler

A \"metadata\" handler is not defined for type \"^0\"

NoDataBuffer

Cannot allocate space for data I/O buffer in container \"^0\"

BadUpdateRead

Incorrect byte length read while reading updates in container \"^0\"

BadUpdateWrite

Write error while writing updates in container \"^0\"

Internal4

Internal error! End-of-updates signal not detected in container \"^0\"

Internal5

Internal error! TOC offset to updates TOC subsection not found in container \"^0\"

NoNewValuesTOC

\"New values\" TOC offset/size missing in TOC object 1 in container \"^0\"

wrappedIDs

Too many objects in container \"^0\" or next user object ID > 0xFFFFFFFF

NoTouchedEntry

Cannot allocate space for recording updating operation in container \"^0\"

NoUpdateObject

Container \"^0\" updating -- cannot find object ID ^1 to be ^2

NoUpdateProperty

Container \"^0\" updating -- cannot find a property ID ^1 in object ID ^2 to be ^3

BadUpdateControl

Container \"^0\" updating -- unknown control byte read (0x^1, during ^2)

NoUpdateType

Container \"^0\" updating -- cannot find value (type ID ^1) in object ^2, property ^3

UndefUpdateObject

Container \"^0\" updating -- cannot find ^1 (ID ^2) to use in ^3

UpdateBadGenNbr

Container \"^0\" updating -- invalid generation number (^1)

BadInsertData

Container \"^0\" updating -- bad updating info (0 segment insert encountered)

BadInsertOffset

Container \"^0\" updating -- insert offset (^1) beyond end of a value

CantRepImmediate

Container \"^0\" updating -- attempt to replace non-immediate with immediate

CantRepBaseType

Container \"^0\" updating -- attempt to replace non-base type value with a base type

CantReference

Value and referenced object not in the same containers (containers \"^0\" and \"^1\")

CM_err_GlobalNameError

Allocation problem while looking up global name \"^0\" in container \"^1\"

CM_err_FlushErr

Error detected in flushing output while closing container ^0

CM_err_CantDelete5

Cannot delete an object with unreleased dynamic values in use in container \"^0\

CM_err_NoTOCBuffer

Cannot allocate space for TOC I/O buffer in container \"^0\"

CM_err_BadTOCCode

Invalid TOC code read (0x^0) from TOC in container \"^1\"

CM_err_TOCoutOfSync

TOC reading out-of-sync (trying to read across buffer boundary) in container \"^0\"

CM_err_TOCParseErr1

TOC parse error - expected object ID, got 0x^0 in container \"^1\"

CM_err_TOCParseErr2

TOC "parse" error - got 0x^0 at an unexpected time in container \"^1\"

CM_err_Unsupported1

8-byte offset in container "^0" are not supported in this implementation

CM_err_CantDelete6

Cannot delete an object currently in use in container \"^1\"

CM_err_AlreadyReleased2

Attempting to release an object already completely released in container \"^0\"

CM_err_BadRefRead

Read error while reading references in container ^0

CM_err_Internal6

Internal error! Missing reference property or value in container \"^0\"

CM_err_ZeroRefKey

Attempt to use a CMReference key of 0 in container \"^0\"

CM_err_NoRefShadowList

Cannot allocate space to record reference in container \"^0\"

Appendix D: Standard Object IDs and Global Names

This entire appendix is new.

Standard Objects

Note that these can also be properties or types, since they are denoted by objects.

```
CM_StdObjID_TOC                          1
```
TOC object ID   (object)

```
CM_StdObjID_TOC_Seed                     2
```
TOC object starting seed   (property)

```
CM_StdObjID_TOC_MinSeed                  3
```
TOC object minimum seed   (property)

```
CM_StdObjID_TOC_Object                   4
```
TOC object   (property)

```
CM_StdObjID_TOC_Container                5
```
TOC object—entire container   (property)

```
CM_StdObjID_TOC_Deleted                  6
```
TOC object amount deleted   (property)

```
CM_StdObjID_TOC_Free                     7
```
TOC object—free space   (property)

```
CM_StdObjID_TOC_NewValuesTOC             8
```
TOC object new values TOC   (property)

```
CM_StdObjID_TOC_Target                   9
```
TOC object—target container ref.   (property)

```
CM_StdObjID_TOC_DeleteList               10
```
TOC object delete update list   (property)

```
CM_StdObjID_TOC_Type                     19
```
TOC object property value type   (type)

```
CM_StdObjID_7BitASCII                    21
```
7-bit ASCII   (type)

```
CM_StdObjID_8BitASCII                    22
```
8-bit ASCII   (type)

```
CM_StdObjID_GlobalTypeName               23
```
Global type name   (property)

```
CM_StdObjID_GlobalPropName               24
```
Global property name   (property)

```
CM_StdObjID_DynamicValues        25
```
    Dynamic values   (property)

```
CM_StdObjID_BaseTypes            26
```
    Base types   (property)

```
CM_StdObjID_BaseTypeList         27
```
    Base type list   (type)

```
CM_StdObjID_TargetContainer      28
```
    Target container   (type)

```
CM_StdObjID_ValueUpdates         29
```
    Value updates for object   (property)

```
CM_StdObjID_UpdatesData          30
```
    Internal updating instruction data   (type)

```
CM_StdObjID_ObjReferences        31
```
    Referenced objects from an object   (property)

```
CM_StdObjID_ObjRefData           32
```
    Reference data   (type)

```
CM_StdObjID_32BitImmValue        40
```
    32-bit immediate data value   (type)

```
CM_StdObjID_16BitImmValue        41
```
    16-bit immediate data value   (type)

```
CM_StdObjID_8BitImmValue         42
```
    8-bit immediate data value   (type)

Global Names

```
CMTOCSeedGlobalName
```
    Apple:Global_Name_TOC_Seed   (property)

```
CMTOCMinSeedGlobalName
```
    Apple:Global_Name_TOC_MinSeed   (property)

```
CMTOCObjectGlobalName
```
    Apple:Global_Name_TOC_Object   (property)

```
CMTOCContainerGlobalName
```
    Apple:Global_Name_TOC_Container   (property)

```
CMTOCDeletedGlobalName
```
    Apple:Global_Name_TOC_Deleted   (property)

```
CMTOCTotalFreeGlobalName
```
    Apple:Global_Name_TOC_Free   (property)

```
CMTOCNewValuesTOCGlobalName
```
    Apple:Global_Name_TOC_NewValuesTOC   (property)
```
CMTOCTargetGlobalName
```
    Apple:Global_Name_TOC_Target   (property)
```
CMTOCDeleteListGlobalName
```
    Apple:Global_Name_TOC_DeleteList   (property)
```
CMTOCValueTypeGlobalName
```
    Apple:Global_Name_TOC_Type   (type)
```
CMDynamicValueGlobalName
```
    Apple:Dynamic_Value   (property)
```
CMBaseTypesGlobalName
```
    Apple:Type_BaseType   (property)
```
CMBaseTypeListGlobalName
```
    Apple:Type_BaseTypeList   (type)
```
CMTargetContainerName
```
    Apple:Target_Container   (type)
```
CMValueUpdatesGlobalName
```
    Apple:Type_ValueUpdates   (property)
```
CMUpdatesDataGlobalName
```
    Apple:Type_UpdatesData   (type)
        "